Centro Universitário de Brasília
Faculdade de Tecnologia e Ciências Sociais Aplicadas

Interface NoSQL integrada a banco relacional para gerenciamento de dados em nuvem privada

Antony Gonçalves Carvalho

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Engenharia da Computação

Orientador
Prof. Msc. Francisco Javier de Obaldía Díaz

Brasília
Novembro, 2014

Antony Gonçalves Carvalho

Interface NoSQL integrada a banco relacional para gerenciamento de dados em nuvem privada

Trabalho apresentado ao Centro Universitário de Brasília (UniCEUB) como pré-requisito para a obtenção de Certificado de Conclusão de Curso de Engenharia de Computação.

Brasília
Novembro, 2014

Antony Gonçalves Carvalho

Interface NoSQL integrada a banco relacional para gerenciamento de dados em nuvem privada

Trabalho apresentado ao Centro Universitário de Brasília (UniCEUB) como pré-requisito para a obtenção de Certificado de Conclusão de Curso de Engenharia de Computação.

Este Trabalho foi julgado adequado para a obtenção do Título de Engenheiro de Computação, e aprovado em sua forma final pela Faculdade de Tecnologia e Ciências Sociais Aplicadas - FATECS.

Prof. Dr. Abiezer Amarília Fernandes
Coordenador do Curso

Banca Examinadora:

Prof. Francisco Javier de Obaldía Díaz, Meste em Engenharia Elétrica.
Orientador

Prof[a]. Layany Zambrano H. Damázio, Meste em Engenharia Elétrica.
UniCEUB

Prof. Marco Antônio de Oliveira Araújo, Mestre em Ciência da Computação.
UniCEUB

Resumo

Enquanto bancos de dados NoSQL são tecnologias fundamentais para *startups* web, as aplicações desenvolvidas neste paradigma podem ter deficiência quanto à consistência, ou facilidades da linguagem SQL, características fortes do modelo relacional. Em contrapartida, este último modelo pode apresentar insuficiências de desempenho como na combinação de dados em tabelas diferentes, falta de flexibilidade no esquema ou dificuldades ao escalar. Atrelado à esta problemática, cresce uma outra concepção, a computação em nuvem, um paradigma de computação orientado a serviços que mudou a forma como a infraestrutura de computação é disponibilizada e utilizada. Com esta motivação, a dualidade lógica relacional e não relacional têm sido repensada de forma a avaliar as vantagens e desvantagens de cada uma. As aplicações que surgirão provavelmente irão se deparar com uma escolha que não terá alguma característica essencial que o outro modelo fornece. Portanto, este trabalho visa promover a coexistência das identidades de cada modelo e oferecer como alternativa uma solução específica oriunda de um problema de cunho prático, usando metodologias da computação, além de apresentar os principais conceitos neste contexto, como gerenciamento de dados em nuvem, arquiteturas de interface e comunicação para web, e comparativos dos modelos de dados NoSQL. Por fim, são apresentadas as considerações finais sobre o tema, destacando desafios e tendências para o desenvolvimento de soluções de bancos de dados híbridos.

Palavras-chave: NoSQL, LightBase, banco de dados híbrido, computação em nuvem

Abstract

While NoSQL databases are key technologies for web startups, applications developed in this paradigm can have disabilities of consistency, or facilities of the SQL language, strong features of the relational model. In contrast, the latter model can present inadequacies of performance as the combination of data in different tables, lack of flexibility in the scheme or difficulty while scale out. Tied to this issue, another conception grows, cloud computing, a paradigm of service-oriented computing that has changed the way computing infrastructure is available and used. With this motivation, the relational and non-relational logic duality have been rethought in order to evaluate the advantages and disadvantages of each. Emerging applications will be faced with a choice that will not have some essential feature that the other model provides. Therefore, this paper aims to promote co-existence of identities of each model and offer an alternative deriving a specific solution to a problem of practical nature using computing methodologies and presents key concepts in this context, as data management in cloud, interface and web communication architectures, and yet a comparative between NoSQL data models. Finally, the concluding remarks on the subject, highlighting trends and challenges for the development of solutions for hybrid databases are presented.

Keywords: NoSQL, LightBase, hybrid database, cloud computing

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Apresentação do Problema

A evolução dos serviços e produtos sob demanda ou *utility computing*, fornecidos por ambientes de computação em nuvem, além da plataforma Web 2.0 têm alavancado o surgimento de tecnologias de bancos de dados relacionais e não relacionais e também o número aplicações com diferentes tipos de negócios.

Entretanto, cada tecnologia de banco de dados possui características específicas, tornando difícil a escolha adequada na hora de desenvolver uma aplicação. O desenvolvedor ou arquiteto precisa então avaliar as vantagens e desvantagens de cada modelo para decidir qual se aplica melhor ao seu caso.

Um primeiro problema então seria a dualidade lógica relacional e não relacional, que não aproveita dos conceitos um do outro, logo os usuários são obrigados a utilizar um extremo, enquanto que um modelo híbrido (que utilize parcialmente a linguagem SQL e um modelo de dados NoSQL) talvez pudesse se ajustar melhor ao modelo da aplicação.

Este problema acontece na empresa onde a solução proposta neste trabalho foi implantada, pois um dos clientes exigiu um banco relacional, por questões contratuais, quando na verdade o modelo não relacional seria uma melhor opção para o modelo da aplicação (solução web para documentos recebidos e expedidos).

Durante o desenvolvimento do trabalho, foi identificado um segundo problema relacionado a alta flexibilidade dos dados do modelo não relacional, problema que pode dificultar o desenvolvimento por aumentar a complexidade de consultas além de gerar inconsistências nos dados.

A proposta deste projeto é resolver esses dois problemas centrais e os problemas a eles correlatos, demonstrando por meio de uma interface de programação de aplicação (API) com características de um modelo de dados não relacional ou NoSQL, integrada um banco de dados relacional, possibilitar o gerenciamento de dados no ambiente da nuvem privada. Além disso, são apresentados os principais conceitos neste contexto, incluindo uma classificação e um comparativo entre as arquiteturas de bancos dados como serviço na nuvem.

Portanto, a questão de pesquisa formulada para este trabalho é: Como é possível implementar um arquitetura de bancos de dados não relacional de modo a poder utilizar funcionalidades do modelo relacional?

1.2 Objetivos do Trabalho

1.2.1 Objetivo Geral

O objetivo deste trabalho é desenvolver uma interface de comunicação (API) que possua características de um modelo de dados não relacional e integrar esta a um banco de dados relacional, de modo que possibilite o gerenciamento de dados no ambiente da nuvem privada.

1.2.2 Objetivos Específicos

Para atingir o objetivo geral, os seguintes objetivos específicos são identificados:

- Analisar a arquitetura da computação em nuvem, assim como os modelos de implantação e serviço.
- Analisar as técnicas de gerenciamento de dados na nuvem para aplicar os conceitos no protótipo.
- Implementar um modelo de dados não relacional integrado a uma interface de comunicação que opere no ambiente de nuvem privada.
- Desenvolver um meio de esquematizar a estrutura dos dados do banco para conter a flexibilidade do modelo não relacional.
- Integrar a interface a um banco relacional de modo a poder utilizar parcialmente a linguagem SQL para consulta aos dados.

1.3 Justificativa e Importância do Trabalho

Sistemas de gerenciamento de banco de dados relacionais (SGBDRs) ainda são amplamente utilizados, dominando mercado e fornecem um conjunto integrado de serviços para uma variedade de requisitos, que incluem principalmente apoio para o processamento de transações, mas também ao processamento analítico e apoio à decisão. Porém, em alguns contextos, este modelo pode não ser a melhor opção(ATZENI; BUGIOTTI; ROSSI, 2014):

- Sua performance pode não ser adequada, principalmente quando a cláusula JOIN é amplamente utilizada, pois pode ser uma operação bastante custosa para o banco.

- A estrutura do modelo relacional, embora sendo eficaz para muitas aplicações tradicionais, é considerada muito rígida para outras.
- Todo o poder do modelo relacional com transações complexas e consultas complexas, não é necessário em alguns contextos, onde "as operações simples"(ler e escrever, envolvendo pequena quantidade de dados) são o suficiente.

Com esta motivação, a dualidade lógica relacional e não relacional têm sido repensada de forma a avaliar as vantagens e desvantagens de cada uma. As Aplicações que surgirão provavelmente irão se deparar com uma escolha que não terá alguma característica essencial que o outro modelo fornece.

Isso pode acontecer por exemplo no desenvolvimento de aplicações Web, onde pode ser vantajoso utilizar diretamente com o banco tanto dados estruturados, bastante difundidos na Web e nos sistemas NoSQL, quanto a linguagem SQL, bastante conhecida e utilizada em aplicações.

Tendo isso em consideração, este trabalho visa desenvolver uma interface que possa oferecer parcialmente a lógica relacional sem perder a funcionalidade oferecida pela lógica não relacional. Como a maioria dos bancos de dados NoSQL do mercado suportam requisitos de gerenciamento de dados na nuvem nativamente, o projeto também deverá herdar tais características. Com esta visão, a proposta apresenta uma opção mais viável para aplicações que precisam de utilidades dos dois mundos.

1.4 Escopo do Trabalho

A ferramenta desenvolvida neste trabalho tem como escopo uma interface de comunicação NoSQL integrada a uma banco de dados relacional e o gerenciamento de dados em nuvem privada. Com gerenciamento entende-se prover ao usuário a recuperação e alteração dos dados.

Não faz parte do escopo o tratamento de segurança da informação transitada no uso da ferramenta nem efetuar o controle de usuários da mesma. Não é objetivo deste trabalho provar que a solução aqui proposta é melhor que o modelo relacional ou NoSQL, mas sim oferecer uma alternativa aos diversos tipos de aplicações.

Como o foco deste projeto é a interface, o ambiente de nuvem privada e o banco relacional serão descritos com menor ênfase, mas com a devida referência para o leitor possa aprofundar mais o assunto se assim desejar.

A figura 1.1 mostra os componentes do projeto. Nela podem ser observados os quatro principais, desconsiderando o contexto da nuvem. A API REST e o modelo de dados foram desenvolvidos, e em conjunto com o servidor WEB, representam a interface NoSQL. O SGBDR, aplicado ao projeto, completa os seus componentes.

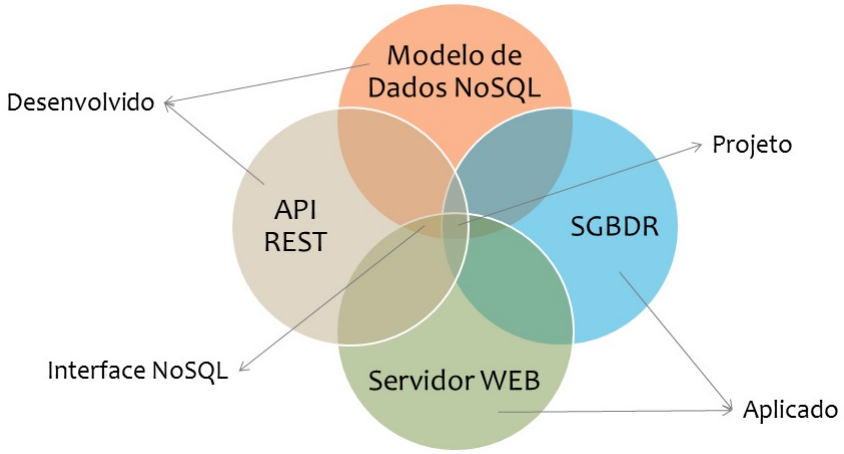

Figura 1.1: Componentes do Projeto

A figura 1.2 mostra o diagrama macro do projeto. Nela podem ser observadas as três principais camadas em que está dividido o projeto: comunicação pela nuvem, processamento pela interface e persistência dos dados. No diagrama também está presente uma representação das aplicações, ou seja, os usuários da ferramenta (solução) descrita por este trabalho. As aplicações provocam o evento requisitando informações do serviço (recuperação ou alteração dos dados). A comunicação é feita através do ambiente de nuvem, que demanda do usuário poucas configurações e administração. A interface recebe a requisição e processa a resposta, comunicando-se com o banco de dados relacional.

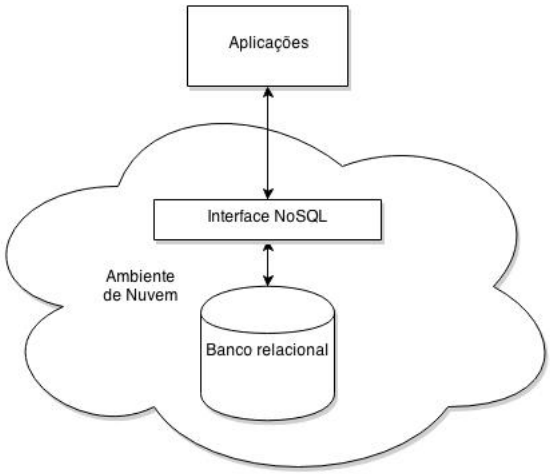

Figura 1.2: Diagrama Macro das Camadas do Projeto

1.5 Resultados Esperados

Com o desenvolvimento do trabalho proposto, pretende-se encontrar um modelo de dados não relacional que se encaixe no modelo relacional de forma que o usuário possa utilizar tanto conceitos do modelo relacional (como o uso parcial da linguagem SQL) como do modelo não relacional.

Espera-se que a ferramenta possa absorver as características de gerenciamento no ambiente da nuvem privada, funcionando como serviço, herdando características essenciais como escalabilidade de processamento.

1.6 Estrutura do Trabalho

Este trabalho está estruturado em 5 capítulos. O Capítulo 1 apresenta o problema, descreve os objetivos gerais e específicos, traz a justificativa do trabalho, delimita o seu escopo e enuncia os resultados esperados. O Capítulo 2 descreve as bases metodológicas para a resolução do problema, e se aprofunda nos aspectos técnicos que são relevantes para a resolução do problema. O Capítulo 3 apresenta o modelo de solução proposto para solução do problema, detalhando sua estrutura e componentes. O capítulo 4 mostra os resultados dos testes do protótipo e uma avaliação de seu desempenho. O Capítulo 5 traz as considerações finais sobre o trabalho, as dificuldades encontradas e as sugestões para trabalhos futuros.

Capítulo 2

Metodologia e referencial teórico

Este capítulo trata dos conceitos necessários para o atingir objetivo do trabalho, visando a resolução do problema proposto na introdução. para isso, são descritas técnicas, métodos, metodologias e outras ferramentas que serão utilizadas na formulação do modelo de resolução do problema a ser proposto no capítulo 3. Também serão apresentados os motivos e justificativas que levaram à escolha de tais métodos e ferramentas no presente trabalho.

Para realizar justificar a escolha por determinada metodologia, será feita uma breve descrição da mesma, de forma a permitir sua compreensão e funcionamento, bem como a maneira pela qual a mesma será empregada na resolução do problema central do trabalho.

Por fim, será abordado uma conclusão, ressaltando os principais pontos e introduzindo ao próximo capítulo.

2.1 Computação em Nuvem

A computação em nuvem pode ser considerada uma tendência recente que surgiu com o objetivo principal da prestação de serviços de tecnologia atendendo a demanda do usuário. Segundo (BUYYA et al., 2009) , é uma metáfora para internet ou infraestrutura de comunicação entre os computadores arquiteturais, baseada na abstração que oculta a complexidade de infraestrutura.

Com a constante evolução da computação, a necessidade de construir infraestruturas complexas, onde configuração, instalação e atualização são feitas se tornou maior. Então estes recursos estão propensos a serem substituídos por plataformas de terceiros, permitindo usuários e empresas utilizarem serviços sob demanda, com independência de localização e transparência de complexidade.

A internet e a computação em nuvem estão intimamente ligados um ao outro. Cada parte desta infraestrutura é provida como um serviço e estes são normalmente alocados em centros de dados, utilizando hardware compartilhado para computação e armazenamento.

Com isso, os usuários estão movendo seus dados e suas aplicações para a nuvem e assim podem acessá-los de forma simples e de qualquer local.

Os centros de dados fornecedores de máquinas usadas para formar um ambiente de computação em nuvem geralmente são compostos de várias máquinas físicas e virtuais, com as mesmas configurações de software, mas pode ter variação na capacidade de hardware em termos de CPU (*Central Processing Unit*), memória e armazenamento em disco (SOROR et al., 2010). A figura 2.1 mostra um modelo de arquitetura em camadas de uma infraestrutura de computação em nuvem.

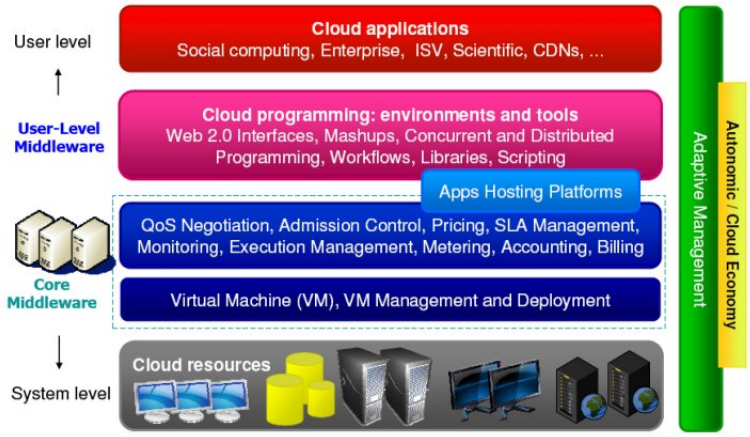

Figura 2.1: Arquitetura da Computação em Nuvem (VECCHIOLA; CHU; BUYYA, 2009)

Nessa figura pode-se observar que as camadas do modelo mudam progressivamente o ponto de vista do sistema para o usuário. A quarta e última camada é caracterizada pelos recursos físicos (*clusters*, *datacenters*, máquinas *desktop*) onde a infraestrutura é montada.

Na terceira camada, um *middleware*, ou *software* mediador central gerencia os recursos físicos de modo a explorá-los da melhor forma possível, usando a técnica da virtualização. Esta camada pode fornecer serviços mais avançados, como QoS (*Quality of Service*), controle de administração, precificação, monitoramento, gerenciamento de execução e gerenciamento SLA (*Service-Level Agreement*).

Juntas, essas duas camadas representam a plataforma onde as aplicações são implantadas na nuvem. Na segunda camada, ou na camada de *middleware* no nível do usuário estão as ferramentas e ambientes de programação para a nuvem, como interfaces Web 2.0, *mashups* (sites ou aplicações web personalizadas que usam conteúdo de várias fontes) e bibliotecas de programação. A proposta da interface, foco deste trabalho localiza-se nesta camada deste modelo arquitetural da computação em nuvem, pois é uma ferramenta para as aplicações na camada de primeiro nível, descritas a seguir.

A camada de primeiro nível trata das aplicações construídas para o ambiente de computação em nuvem, tais como de computação social, empresariais e científicas.

2.2 Modelos de Serviço da Computação em Nuvem

Os modelos de serviço são padrões arquiteturais para soluções de computação em nuvem. Dentre os modelos de serviço do ambiente de computação em nuvem, três são identificados (MELL; GRANCE, 2011):

- *Software como Serviço (SaaS)*: O usuário pode usar a aplicação do provedor implantada na infraestrutura de nuvem. Neste modelo o usuário não controla a infraestrutura incluindo servidores, rede, sistemas operacionais ou armazenamento, exceto nos casos em que a própria aplicação forneça este tipo de serviço. Como exemplos de SaaS podemos destacar os serviços de *Customer Relationship Management* (CRM) da Salesforce (SALESFORCE, 2014) e o *Google Docs* (GOOGLE, 2014b).
- *Plataforma como Serviço (PaaS)*: O usuário pode implantar infraestrutura de aplicações na nuvem usando linguagens de programação, bibliotecas, serviços ou ferramentas suportadas pelo provedor. Neste modelo o usuário não controla a infraestrutura incluindo servidores, rede, sistemas operacionais ou armazenamento, mas tem controle sobre as aplicações implantadas e configurações do ambiente hospedeiro das aplicações. *Google App Engine* (GOOGLE, 2014a) e *Heroku* (HEROKU, 2014) são exemplos de PaaS.
- *Infraestrutura como Serviço (IaaS)*: O usuário pode provisionar processamento, servidores, rede, ou armazenamento e outros recursos computacionais fundamentais onde pode implantar *softwares* arbitrários, como sistemas operacionais e aplicações. Porém, neste modelo o usuário não controla a infra-estrutura da nuvem. O *Amazon Elastic Cloud Computing* (EC2) (AMAZON, 2014), *Microsoft Azure* (MICROSOFT, 2014a) e *OpenStack* (OPENSTACK, 2014) são exemplos de IaaS.

Neste trabalho, o protótipo caracteriza-se com o modelo de serviço SaaS, pois a infraestrutura de rede e servidores não poderá ser controlada pelo usuário, apenas os dados intermediados pela interface implantada pelo provedor na infraestrutura de nuvem.

2.3 Modelos de Implantação da Computação em Nuvem

Os modelos de implantação tratam da disponibilidade dos ambientes de computação em nuvem. Fatores como o processo de negócio e o tipo de informação podem originar o acesso ou restrição de recursos. Quanto aos modelos de implantação, quatro são identificados (MELL; GRANCE, 2011):

- *Nuvem privada*: uma empresa ou terceiros administram a infra-estrutura da nuvem utilizada exclusivamente por uma organização, ou seja, é uma nuvem local ou remota.

- *Nuvem pública*: qualquer usuário que conheça a localização do serviço pode utilizá-lo, ou seja, a infraestrutura de nuvem é disponibilizada para o público em geral.
- *Nuvem comunidade*: fornece uma infraestrutura compartilhada por uma comunidade de organizações com interesses em comum.
- *Nuvem híbrida*: a infraestrutura pode ser uma composição de um ou mais tipos de modelos de implantação de nuvem, conectadas por meio de tecnologia padronizada.

Este trabalho pretende fazer uma implementação em nuvem privada, portanto o serviço não será compartilhado com usuários em escopo global, logo toda avaliação será feita levando em consideração a utilização local.

2.4 *Representational State Transfer* (REST)

REST é um estilo arquitetural, não um protocolo ou uma implementação. Segundo o idealizador da arquitetura REST, a criação da arquitetura REST é a composição de vários estilos de arquitetura para a criação de um novo estilo.

O REST é um estilo híbrido derivado de vários estilos arquitetônicos baseados em rede, combinados com restrições que definem uma interface como conector uniforme, REST é por sua vez, um estilo arquitetural que é direcionado para sistemas de hipermídia distribuídos (FIELDING; TAYLOR, 2002).

O conceito por trás do REST é simplificar a conexão e comunicação entre máquinas, de maneira que o protocolo HTTP (*Hypertext Transfer Protocol*), um protocolo para transferência de hipertexto usado para comunicação na camada de aplicação do modelo OSI (*Open Systems Interconnection*) seja usado para fazer requisições e chamadas entre máquinas.

O REST se destaca pela simplicidade em relação a outros mecanismos mais complexos de comunicação como o CORBA (*Common Object Request Broker Architecture*) e o SOAP (*Simple Object Access Protocol*). A arquitetura REST tem alguns princípios fundamentais, ilustrados na figura 2.2.

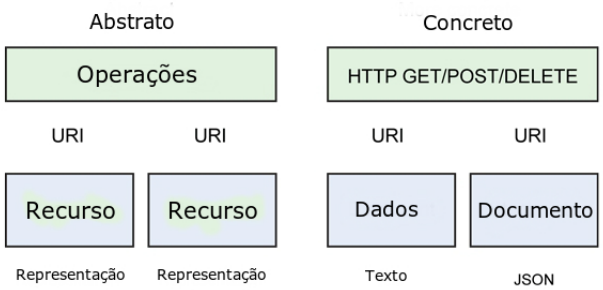

Figura 2.2: Visão de Alto Nível do REST (JONES, 2012)

De acordo com a figura, existem três princípios: operações, identificadores uniformes e recursos. Em uma visão mais concreta, no centro de uma arquitetura REST estão um conjunto de recursos. Estes recursos são identificados por URIs (*Uniform Resource Identifier*) e uma representação interna, geralmente uma forma de dado auto-descrita como JSON (*JavaScript Object Notation*). E finalmente um conjunto de operações para manipulação dos recursos utilizando os métodos padrão do protocolo HTTP (GET, POST, PUT e DELETE).

REST define uma arquitetura cliente-servidor na qual os clientes não tem acesso direto ao recurso, mas uma representação do recurso através de uma interface uniforme. Como muitas arquiteturas cliente-servidor, REST é implementado com camadas como mostra a figura 2.3, permitindo a exploração das capacidades que as camadas inferiores (balanceamento de carga por HTTP por exemplo) fornecem.

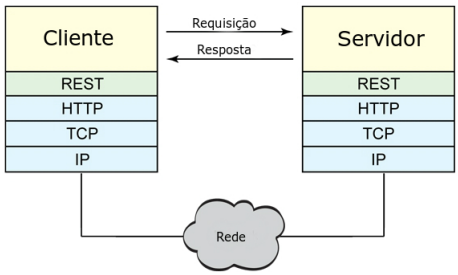

Figura 2.3: Arquitetura em Camadas de Interações REST (JONES, 2012)

Como mostrado na figura, o REST opera como uma camada entre o cliente ou servidor e o protocolo HTTP. Neste trabalho, a arquitetura REST será usada em conjunto com uma API, discutida mais detalhadamente na seção 2.5.

2.5 *Application Programming Interface* (API)

API é uma especificação destinada a ser usada como uma interface por componentes de *software* para se comunicarem uns com os outros. Uma API pode incluir especificações para rotinas, estruturas de dados, objetos de classes e variáveis (AN, 2011).

Em linguagens orientadas a objeto, uma API geralmente inclui uma descrição de um conjunto de definições de classe, com um conjunto de comportamentos associados a essas classes. Este conceito abstrato é associado com funcionalidades reais, ou disponibilizado pelas classes que são implementadas em termos de métodos de classe ou, mais geralmente por todos os seus componentes públicos, incluindo qualquer entidade interna tornada pública, como campos, constantes, e objetos aninhados.

Uma API pode ser dependente de linguagem de programação, o que significa que será apenas acessível com uso da sintaxe e elementos de uma linguagem específica, ou pode ser independente, o que significa que foi escrita de modo que pode ser chamada por várias linguagens.

No contexto da Web, enquanto "API Web"é sinônimo de serviço Web ou Web 2.0, a arquitetura REST têm sido utilizada para permitir a comunicação entre os nós da rede. A prática de publicação de APIs permitiu comunidades da Web criarem uma arquitetura aberta para compartilhamento de conteúdo e dados entre comunidades e aplicações. Desta forma, o conteúdo que é criado em um lugar pode ser dinamicamente publicado e atualizado em vários locais na web (AN, 2011).

Existem três componentes distintos envolvidos em um projeto de uma API usando a arquitetura REST: a aplicação, o código da API, e o cliente. A figura 2.4 ilustra como estes três componentes interagem.

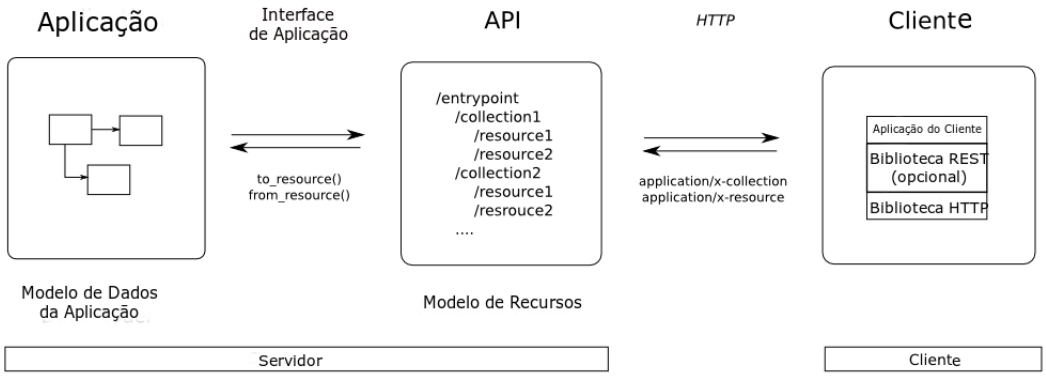

Figura 2.4: RESTful API *design* (GEERT, 2010)

Como na figura, a aplicação possui um modelo de dados que será transformado para sua representação em recurso ou vice-versa. A API faz o tratamento do modelo de recursos enquanto o REST se encarrega de realizar a comunicação com o cliente usando o protocolo HTTP. Desta forma qualquer linguagem de programação que seja capaz de se comunicar pela rede utilizando o protocolo HTTP poderá ser usada pelo cliente neste modelo. Este trabalho busca reproduzir este modelo de interação para permitir a comunicação com os clientes.

2.6 Servidor Web

Um servidor web é um processo responsável por disponibilizar páginas web e outros recursos a um ou mais clientes. Quando um cliente solicita um serviço ao servidor, este processa a requisição e empacota os resultados em uma mensagem de resposta, que é enviada ao cliente (TANENBAUM; STEEN, 2007). Um servidor web normalmente recebe e atende requisições do cliente por meio do protocolo HTTP.

Neste trabalho, o servidor Apache (APACHE, 2014) um dos servidores web mais popular e bem sucedido do mundo (ALECRIM, 2010), foi usado na implementação da

comunicação de uma API NoSQL (em conjunto com a arquitetura REST) com o cliente pelo ambiente de nuvem privada. Dentre as vantagens do Apache que levaram a sua escolha pode-se destacar o fato de possuir compatibilidade com diversas plataformas, além de ser um software livre (o que possibilita que ele seja melhorado por diversas pessoas com o passar dos anos).

2.7 Gerenciamento de Dados em Nuvem

As empresas tendem a reduzir o custo total por meio da utilização de infraestrutura e sistemas de terceiros, e podem atingir este objetivo utilizando SGBDs em ambientes de computação em nuvem (ABADI, 2009). Neste contexto, diversos sistemas e arquiteturas estão sendo desenvolvidos para suprir as novas demandas de aplicações com diferentes requisitos de processamento e armazenamento de dados (GROLINGER et al., 2013).

Porém esta proliferação de diferentes abordagens traz agora outros problemas como escolha correta para a o tipo de aplicação, implantação e migração de dados, além de problemas relacionados com gerenciamentos de dados na nuvem. Por esses motivos, alguns SGBDs estão sendo disponibilizados como serviço na nuvem, encapsulando e ocultando a complexidade de gerenciamento, oferecendo acesso simples além de garantias de qualidade de serviço.

O ambiente de gerenciamento na nuvem tende a ser constituído por recursos homogêneos por poder facilitar a sua administração, porém mais recentemente, ambientes heterogêneos também são comumente utilizados. O acesso de dados é padronizado de forma a simplificar o acesso aos dados, fornecendo APIs simples, utilizando a linguagem SQL ou variações, além de fornecerem meios de atualização de forma concorrente, e com suporte a transações, podendo estas ser do tipo ACID ou variações.

Segundo (MACHADO, 2014), algumas técnicas do gerenciamento de dados em ambientes de computação em nuvem são diferenciados das técnicas tradicionais. Dentre estas técnicas, podemos citar bancos de dados como serviço, virtualização, transações, escalabilidade, elasticidade, armazenamento e processamento de consultas. Esta técnicas são apresentadas a seguir.

2.7.1 *Data as a Service*(DaaS)

Com os sistemas de bancos de dados como serviço, ou *Data as a Service*(DaaS), o prestador hospeda um banco de dados e o fornece como serviço. Cada inquilino contrata os serviços fornecidos pelo provedor. Este provedor então mantem o serviço para vários inquilinos em centros de dados. Isso pode ser feito utilizando uma implantação de banco de dados multi-inquilino.

Neste contexto, uma implementação de bancos de dados multi-inquilino pode ser usado para prover serviços de bancos de dados na nuvem. Banco de dados multi-inquilino é uma otimização para serviços hospedados onde vários usuários são consolidados no mesmo sistema operacional (JACOBS; AULBACH, 2007).

De acordo com (TAYLOR; GUO, 2007), há vários níveis de isolamento de dados para aplicações do modelo de serviço SaaS que vão desde um ambiente isolado até um ambiente totalmente compartilhado. A figura 2.5 exibe as abordagens de isolamento em ambientes de dados. As implementações ao longo deste espectro incluem os seguintes ambientes:

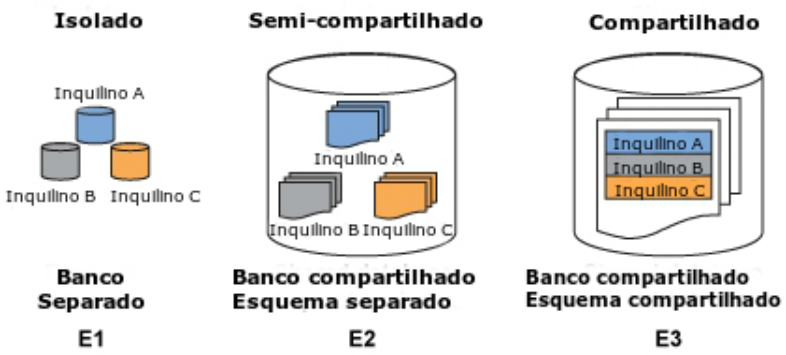

Figura 2.5: Ambientes de Dados Isolados vs. Compartilhados (TAYLOR; GUO, 2007)

- *Totalmente isolado*: no ambiente isolado E1, os bancos de dados são separados por inquilinos, onde cada inquilino possui o seu banco de dados. Este modelo oferece mais flexibilidade, em termos de configuração, pois as tabelas são desenhadas com colunas customizadas para suportar um único inquilino. Esta abordagem pode apresentar um alto custo de manutenção, pois é necessário para o provedor gerenciar diversas instâncias de banco de dados. Em termos de escalabilidade, o número de instâncias de banco de dados cresce linearmente com o número de inquilinos.
- *Parcialmente compartilhado*: no ambiente parcialmente compartilhado E2, os bancos de dados são compartilhados, havendo uma separação por esquemas, onde cada inquilino possui o seu esquema. Este modelo oferece um nível menor de isolamento dos dados que E1 e também oferece a mesma facilidade de configuração que E1, com colunas customizadas. Esta abordagem diminui o custo de manutenção de gerenciamento de diferentes instâncias de bancos de dados e a escalabilidade é limitada ao número de esquemas suportados pelo SGBD e memória disponível.
- *Totalmente compartilhado*: no ambiente compartilhado E3, os bancos de dados e os esquemas são compartilhados, podendo haver separação por tabelas, onde cada inquilino possui a sua tabela. Este modelo é visto como o mais difícil de configurar. Nesta abordagem, o provedor de serviço mantém apenas uma instância de banco de dados simples, o que reduz o custo de manutenção. A escalabilidade nesta abordagem é facilitada, visto que o número de tabelas é determinado pelo esquema independentemente do número de inquilinos.

Para o trabalho em questão será utilizada a abordagem do ambiente totalmente compartilhado, para beneficiar questões de facilidade de manutenção e escalabilidade a serem implementadas futuramente. Será mantida apenas uma instância de banco de dados, em que cada inquilino pode ter uma ou mais tabelas.

2.7.2 Virtualização

A virtualização (SOROR et al., 2010) também vem se consolidando no paradigma de gerenciamento de dados em nuvem, pois esta técnica permite o provisionamento de recursos de forma compartilhada, por meio de simulação de uma plataforma de hardware, podendo facilitar a administração do parque computacional. Com esta técnica também é possível que várias aplicações funcionem com recursos dedicados, permitindo que sejam aproveitados mais eficientemente em termos de capacidade de armazenamento e processamento, por exemplo.

Com a virtualização, proporciona-se o compartilhamento da infraestrutura para os vários inquilinos inseridos no contexto. Um inquilino então pode ser desde um usuário acessando uma aplicação que acessa um SGBD até um SGBD implantado em uma infraestrutura. A tabela 2.1 mostra uma classificação do isolamento de recursos em cada modelo de serviço.

Modo de Compartilhamento	**Isolamento**	**IaaS**	**PaaS**	**SaaS**
1.Hardware	VM	*x*		
2.Máquina Virtual	Usuário SO		*x*	
3.Sistema Operacional	Instância do BD		*x*	
4.Instância	BD		*x*	
5.Banco de Dados	Esquema		*x*	
6.Tabela	Linha			*x*

Tabela 2.1: Compartilhamento de Recursos nos Modelos de Serviço (ELMORE et al., 2011)

Nos primeiros níveis ou nos níveis mais externos, há compartilhamento de recursos na mesma máquina física, por meio de várias máquinas virtuais (VMs). Estas máquinas podem compartilhar sistemas operacionais, contas de usuário e até instâncias de SGBDs.

Nos últimos níveis ou nos níveis mais internos, que são mais amplamente utilizados, há compartilhamento de instâncias (processos) de bancos de dados, esquemas e linhas de tabelas. Estes níveis podem apresentar interferência entre os inquilinos, e como consequência, degradar o desempenho do sistema.

2.7.3 Transações

É de suma importância nos sistemas de bancos de dados distribuídos e centralizados os conceitos de transações pois definem o nível de consistência e integridade dos dados nestes ambientes. A utilização de transações distribuídas define o controle do processamento de dados em todo ambiente de computação em nuvem e tem a responsabilidade de garantir as propriedades transacionais (MACHADO, 2014). Antes de citar os tipos de transações é relevante citar o teorema CAP (*Consistency, Availability, Partition Tolerance*), que mostra que os sistemas distribuídos não podem assegurar as seguintes propriedades simultaneamente (BREWER, 2012):

- *Consistência*: todos os nós tem a mesma visão dos dados ao mesmo tempo.
- *Disponibilidade*: falhas em nós não impedem os demais nós de continuar a operar.
- *Tolerância de particionamento*: o sistema continua a operar mesmo com a perda arbitrária de dados.

De acordo com o teorema, um sistema distribuído pode suportar apenas duas dessas três propriedades ao mesmo tempo. Tendo isso em vista, outras abordagens para o gerenciamento de dados usam diferentes formas de transação. Uma forma comumente usada nos SGBDR é a ACID (*Atomicity, Consistency, Isolation, Durability*) (PRITCHETT, 2008):

- *Atomicidade*: a transação será executada totalmente ou não será executada.
- *Consistência*: garante que o banco passará de um estado consistente para outra forma consistente.
- *Isolamento*: garante que uma transação não será interferida por nenhuma outra transação concorrente.
- *Durabilidade*: garante que aquilo que foi persistido não será mais perdido.

Porém para os sistemas distribuídos na nuvem, obedecer o teorema CAP enquanto suporta propriedades ACID não é uma tarefa fácil, e por isso a consistência e o isolamento são geralmente sacrificados, resultando na abordagem BASE (*Basic Availability, Soft-state, Eventual consistency*) (PRITCHETT, 2008):

- *Basicamente disponível*: o sistema estará disponível basicamente todo o tempo.
- *Estado leve*: o estado do sistema não precisa estar sempre consistente.
- *Eventualmente consistente*: o sistema torna-se consistente em um determinado momento.

Como a proposta do trabalho é integrar uma interface a um banco relacional, as transações ACID serão herdadas da camada mais inferior, logo a interface terá como característica consistência forte.

2.7.4 Escalabilidade

A escalabilidade está relacionada a modificação nos componentes de um sistema para atender a área de um problema. Embora a escalabilidade não esteja contida na velocidade ou no desempenho de um sistema, geralmente dentre as áreas dos problemas resolvidos utilizando técnicas de escalabilidade estão estes dois conceitos. São identificados dois tipos de técnicas para escalabilidade em sistemas (GREGOL, 2012), horizontal e vertical.

Entende-se por escalabilidade vertical (*scale-up*) o aumento do processamento pelo *hardware* de servidores individuais, com a adição de recursos como memória, processadores ou disco rígido mais rápido para atendimento de uma crescente demanda de requisições em uma aplicação. Esta técnica pode funcionar bem para determinadas aplicações, porém as máquinas têm seu limite de *hardware* e o custo para a aquisição de equipamentos de maior capacidade pode ser alto. A figura 2.6 mostra um exemplo de escalabilidade vertical, onde o *hardware* é incrementado (seta para cima) para uma maior capacidade.

Figura 2.6: Esquema de Escalabilidade Vertical (GOGRID, 2010)

Com a escalabilidade horizontal (*scale-out*), a adição de nós (novos servidores e sistemas de *software*) na arquitetura de um sistema permite a distribuição do trabalho a ser realizado por múltiplas máquinas que podem compartilhar o mesmo recurso e servir uma mesma aplicação. Esta técnica pode oferecer maior flexibilidade mas necessita de um planejamento mais detalhado e específico. A figura 2.7 mostra um exemplo de escalabilidade horizontal, onde o *hardware* é adicionado (seta para o lado) para um número maior de elementos.

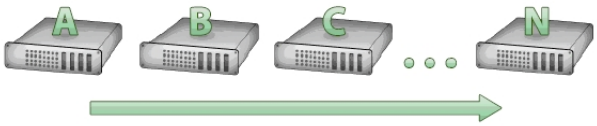

Figura 2.7: Esquema de Escalabilidade Horizontal (GOGRID, 2010)

É importante ressaltar que para o trabalho em questão, é possível implementar tanto escalabilidade relativa ao armazenamento dos dados ou ao processamento de consultas. Este trabalho não se concentra em fornecer escalabilidade de armazenamento, visto que isto pode ser feito utilizando técnicas de escalabilidade para bancos de dados relacionais, o que não faz parte dos objetivos do projeto. Caso o leitor queira se aprofundar, existem pesquisas relacionadas ao assunto (CHEN; HSU; WU, 2012), (LOWERY, 2003). Porém, a escalabilidade de processamento será tratada neste trabalho de forma horizontal, com a adição de um ou mais nós contendo a lógica da interface implantada.

2.7.5 Elasticidade

Termos como escalabilidade e elasticidade não podem ser confundidos entre si. A escalabilidade discutida na seção 2.7.4 tem relação com a capacidade do sistema ser expandido a medida do necessário, enquanto a elasticidade pressupõe a capacidade dos recursos se

ajustarem à carga necessária, incluindo aumento ou diminuição dos mesmos (COUTINHO et al., 2013).

Vários autores definem elasticidade, porém de forma geral se dá a ideia da habilidade de criar um número variável de instâncias de máquinas virtuais que dependem da demanda da aplicação. Alguns citam também a forma com que este provisionamento é feito, automaticamente e rapidamente. De acordo com (SHARMA et al., 2011), duas formas principais são utilizadas para implementação da elasticidade: replicação e redimensionamento de recursos.

A replicação pode ser utilizada para implementar a elasticidade com a criação de réplicas para servirem de recursos para a carga de trabalho atual. O redimensionamento de recursos pode permitir o ajustamento da quantidade de recursos de acordo com a carga de trabalho, com o incremento ou decremento de réplicas do ambiente.

Este trabalho utilizará réplicas de máquinas virtuais para poder configurar mais ou menos instâncias com lógica da interface, não se empenhando em resolver questões relacionadas ao redimensionamento de recursos.

2.7.6 Armazenamento e Processamento de Consultas

Os sistemas de gerenciamento de bancos de dados relacionais (SGBDR) tradicionais foram desenhados em uma era em que hardware e software eram diferentes, e agora encontram prolemas para encarar a performance e os requisitos da computação em nuvem. Os sistemas de armazenamento NoSQL se apresentam como alternativas para poder suprir esses novos desafios da computação (GROLINGER et al., 2013).

As técnicas de armazenamento e processamento de consultas utilizadas nos sistemas NoSQL podem se diferenciar bastante das utilizadas nos SGBDR, e como faz parte do escopo deste trabalho integrar uma interface com estas características, é necessário discutir com mais detalhamento estas técnicas. A seção 2.8 traz os conceitos relacionados aos sistemas NoSQL e assuntos a eles correlatos.

2.8 NoSQL

NoSQL ou *Not Only SQL* é um termo usado como um termo genérico para todos os bancos e armazenamento de dados que não seguem o popular e bem-estabelecido princípio do SGBDR e muitas vezes se relacionam com grandes conjuntos de dados acessados e manipulados na escala da Web. Isso significa que NoSQL não é um único produto ou até mesmo uma única tecnologia. Representa uma classe de produtos e uma coleção de diversas e, por vezes relacionada, conceitos sobre armazenamento e manipulação de dados (TIWARI, 2011).

Em um nível genérico, o objetivo de um sistema de gerenciamento de dados para a nuvem é sustentar desempenho e disponibilidade ao longo de um grande conjunto de dados, sem significativo excesso de provisionamento (AGRAWAL et al., 2010). No contexto de

computação em nuvem, os sistemas de armazenamento de dados NoSQL comportam bem os requisitos necessários, e vem sendo discutidos pela comunidade de pesquisa.

Além do conceito da da arquitetura NoSQL, ao longo desta seção serão apresentadas algumas das motivações para o seu surgimento, suas características, alguns dos modelos de armazenamento de dados mais utilizados e um breve comparativo entre esses modelos.

2.8.1 Motivação para o surgimento do NoSQL

Enquanto SGBDRs têm sido a solução padrão para o gerenciamento de dados em ação por muitos anos, alguns aplicativos modernos sofreram mudanças de requisitos para os quais SGBDRs têm sido insatisfatórios em vários aspectos (STRAUCH; SITES; KRIHA, 2011). Alguns dos principais motivos por trás do movimento NoSQL foram (NÄSHOLM, 2012):

- *Vazão lenta*: Algumas aplicações web exigem que o armazenamento de dados processe volumes maiores do que a maioria SGBDRs seria capaz de manusear.
- *Não foram construídos para escalar horizontalmente*: SGBDRs foram originalmente construídos para escalar verticalmente. No entanto, a escala vertical tem suas limitações no que há um limite superior sobre o quanto de hardware que pode ser adicionado e eficientemente utilizada pelo software. Hoje, alguns SGBDRs oferecem escalabilidade horizontal, mas esta abordagem geralmente não é utilizada.
- *Mapeamento objeto-relacional é caro*: Hoje, a programação orientada a objetos é o paradigma de programação mais prevalente. Uma aplicação persistindo objetos em um SGBDR requer um mapeamento entre o modelo do objeto e do modelo relacional. Definir isso toma tempo e processamento.
- *O pensamento "One Size Fits All" é falho*: Este pensamento quer dizer que os SGBDRs são vistos como um instrumento geral que pode lidar com todos os requisitos que uma aplicação pode ter no gerenciamento de dados. Uma vez que diferentes aplicações podem ter diferentes exigências sobre a consistência, desempenho e assim por diante, este pensamento pode ser falho.
- *ACID não é sempre necessário*: Em algumas aplicações, transações ACID (veja seção 2.7.3) não são necessárias, o que significa que pode se perder alguns *trade-offs* como desempenho.

2.8.2 Características do Armazenamento de Dados NoSQL

Segundo (NÄSHOLM, 2012), a maioria dos bancos de dados NoSQL compartilham um conjunto comum de características. Como NoSQL é um conceito amplo, estas características podem ter exceções. Ainda assim, podem servir para dar uma idéia geral sobre o que bancos de dados NoSQL são:

- *Distribuídos*: Bancos de dados NoSQL são geralmente sistemas distribuídos onde várias máquinas cooperam com processamento e armazenamento. Os dados são replicados em vários servidores para redundância e alta disponibilidade.
- *Escalabilidade horizontal*: Não há limites (realísticos) superiores para números de nós de um *cluster*, onde as máquinas podem ser adicionadas e removidas dinamicamente.
- *Grandes volumes*: Muitos sistemas NoSQL foram construídos para armazenar quantidades de dados muito grandes em pouco tempo.
- *BASE ao invés de ACID*: Para um número crescente de aplicações, disponibilidade e tolerância de particionamento são fatores importantes. Construir uma base de dados com estes, proporcionando propriedades ACID é difícil, e é por isso que consistência e isolamento muitas vezes são recusadas, resultando na chamada abordagem BASE (veja seção 2.7.3).
- *Modelos de dados não relacionais*: Os modelos de dados variam, mas em geral não são relacionais. Normalmente, permitem desde estruturas simples até as mais complexas e não são tão rígidas como o modelo relacional.
- *Sem definições de esquema*: Os clientes podem armazenar dados como eles desejam, sem ter que aderir a uma estrutura pré-definida, diferentemente dos esquemas explícitos dos bancos relacionais.
- *SQL não é suportado*: Enquanto a maioria dos SGBDRs apoiam algum dialeto SQL, os variantes NoSQL geralmente não o fazem. Em vez disso, cada sistema tem sua própria interface de consulta. Porém há tentativas de unificar as interfaces de consulta de NoSQL (JACKSON, 2011).

2.8.3 Modelos de dados NoSQL

Com as diferentes abordagens das empresas para encontrar um modelo de dados que encaixe nas suas necessidades, alguns modelos de dados, qualquer um que não se aplique ao modelo relacional, têm sido desenvolvidos para suprir requisitos específicos de desempenho, escalabilidade e manutenção. Nesta seção será discutido de forma geral alguns desses modelos.

2.8.3.1 Modelo de Dados Chave-Valor

Um modelo de dados bastante popular é o armazenamento de chaves associados a valores. Os bancos de dados que utilizam este modelo geralmente comportam um volume muito grande de dados, pois sua complexidade é de baixo nível.

Um *HashMap* ou uma matriz associativa é a estrutura de dados mais simples que pode conter um conjunto de pares chave-valor. Essas estruturas de dados são extremamente populares porque proporcionam um algoritmo de complexidade de tempo médio O(1)

muito eficiente, para acessar os dados. A chave de um par chave-valor é um valor único em no conjunto e pode ser facilmente encontrado para acessar os dados (TIWARI, 2011).

Alguns bancos de dados chave-valor podem guardar dados em memória em favor da performance de acesso aos dados, enquanto outros podem persistir os dados em discos, facilitando escalabilidade.

A figura 2.8 mostra um *HashMap* chave-valor operando juntamente com um índice de árvore binária, uma estrutura de dados baseada em nós, onde os dados são organizados de forma a permitir busca binária (COMER, 1979).

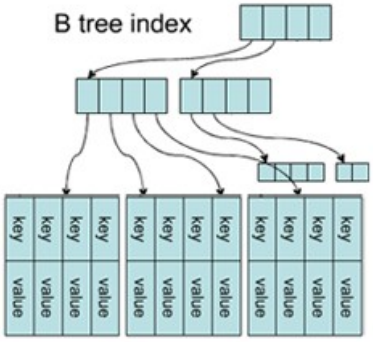

Figura 2.8: Banco Chave-Valor e Índice de Árvore Binária (GRIGORIK, 2009)

2.8.3.2 Modelo de Dados Orientado a Colunas

Sistemas deste tipo podem comportar grandes quantidades de dados, e persisti-los com um certo nível de complexidade.

Os sistemas que utilizam este modelo são inspirados na arquitetura do *Google BigTable* (CHANG et al., 2008). Neste modelo, cada unidade de dados pode ser considerada como um conjunto de pares de chave-valor, em que a unidade em si é identificada com a ajuda de um identificador primário, frequentemente referido como a chave primária. Esta chave primária é chamada *row-key*. Além disso, as unidades são armazenados de forma ordenada. As unidades de dados são classificadas e ordenadas com base na *row-key* (TIWARI, 2011).

De forma geral, uma chave identifica uma linha, que contém dados armazenados em uma ou mais famílias de coluna (FCs). Dentro de uma FC, cada linha pode conter várias colunas (MARCUS, 2011).

Deste modo, o armazenamento orientado a colunas permite que dados sejam armazenados eficazmente. O modelo evita consumir espaço ao armazenar valores nulos por simplesmente não armazenar uma coluna quando um valor não existe para essa coluna (TIWARI, 2011). A figura 2.9 mostra uma tabela do HBase, um banco de dados orientado a colunas.

User			
	Account	Personal	Friends
1001	username = “bob1987” password = “thisisapassword”	firstName = “Bob” lastName = “Smith” ssn = “4hfe94”	2004:firstName = “Alice” 2004:lastName = “Smith” 2004:email = “alice@gmail.com” 1714:firstName = “Charlie” ...
2004	username = “alice”	...	...
1714	...	...	...

Figura 2.9: Modelo de Dados Colunar (ATZENI; BUGIOTTI; ROSSI, 2014)

A figura mostra uma tabela de usuários do HBase. As FCs são enfatizadas no layout: temos *account*, *personal* e *friends*. A FC *account* armazena informações da conta do usuário com colunas como nome de usuário e senha. A FC *personal* armazena informações pessoais, enquanto a FC *friends* armazena dados de relações de amigos. Cada amigo está associado com um número de colunas, os quais tem o identificador do usuário como prefixo, e armazena informações de contato como nome e email. A coluna de nomes na FC *friends* mostra uma prática comum de manuseio de conjuntos de valores com referências a outras linhas.

2.8.3.3 Modelo de Dados Orientado a Documentos

A palavra *documento* em *banco de dados documental* conota conjuntos de estruturas vagamente estruturados de pares chave-valor tipicamente no formato JSON (*Javascript Object Notation* (JSON, 2014)) e não documentos como planilhas (embora estes também possam ser utilizados) por exemplo (TIWARI, 2011). Além disso, o modelo documental é considerado o próximo passo do modelo chave-valor por armazenar estruturas mais complexas com aninhamento ou valores escalares. Geralmente os nomes dos atributos são definidos dinamicamente para cada documento em tempo de execução (GAJENDRAN, 2012).

Os bancos de dados documentais tratam documentos com um todo evitando dividir sua constituição de pares chave-valor. Neste modelo de dados é possível indexar os documentos não só no seu identificador primário, mas também em suas propriedades (TIWARI, 2011).

Em comparação com o modelo relacional, este modelo pode conter todos os dados em uma linha que estendem a 20 tabelas relacionais e agregá-los a um único documento/objeto como mostra a figura 2.10. Isso pode levar a duplicação de dados mas partindo de que o armazenamento não é mais um problema, o resultado é uma melhora na performance, facilidade na distribuição dos dados e no *trade-off* (uma expressão que define uma situação em que há conflito de escolha) para aplicações baseadas na web (COUCHBASE, 2014).

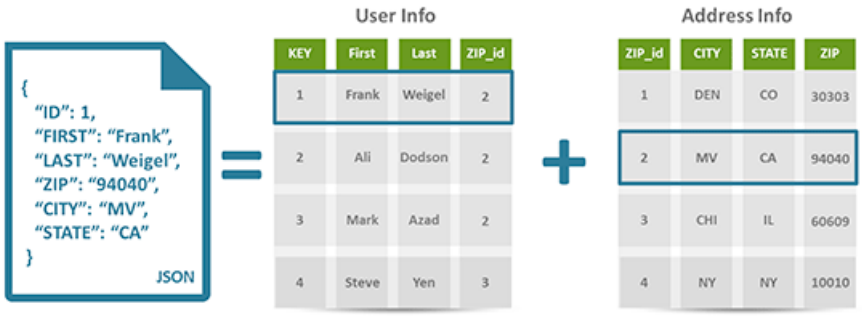

Figura 2.10: Modelo de Dados Documental (COUCHBASE, 2014)

Internamente, a arquitetura de um banco de dados documental geralmente absorve conceitos de hierarquia. A documentação do MongoDB (MONGODB, 2014) mostra a sua implementação, que abriga uma série de bancos de dados.

> "Um banco de dados contém um conjunto de coleções. Uma coleção contém um conjunto de documentos. Um documento é um conjunto de pares de valores-chave. Os documentos têm esquema dinâmico. Esquema dinâmico significa que os documentos na mesma coleção não precisam ter o mesmo conjunto de campos ou estrutura, e campos comuns em documentos de uma coleção podem conter diferentes tipos de dados."

O esquema dinâmico citado, dependendo do tipo de aplicação desenvolvida, pode se tornar um problema. CouchDB (COUCHDB, 2014) e Riak (RIAK, 2014) deixam o rastreamento do tipo para o desenvolvedor. A liberdade e a complexidade dos repositórios de documentos tem vantagens e desvantagens: os desenvolvedores de aplicativos têm muita liberdade na modelagem de seus documentos, mas a lógica de consulta à base de dados pode tornar-se extremamente complexa (MARCUS, 2011). Além disso, em alguns cenários a dinamicidade do esquema documental pode não ser interessante, como nos casos onde o modelo estrutural de dados é conhecido. Para estes tipos de aplicações, as opções de bancos documentais do mercado tornam-se poucas ou nulas.

2.8.3.4 Modelo de Dados Orientado a Grafos

Outra classe de armazenamento NoSQL é armazenamento de grafos. Este tipo de modelo pode representar grandes ou até maiores complexidades de dados que os SGBDRs, porém apresentam desvantagens em relação ao tamanho da bases de dados de um banco chave-valor, por exemplo.

Um grafo é uma estrutura de dados composta de arestas e vértices. A tecnologia de banco de dados orientado a grafos é uma ferramenta eficaz para a modelagem dados quando o foco sobre a relação entre as entidades é uma força motriz no desenho de um modelo de dados. Modelagem de objetos e as relações entre eles significa quase que tudo

pode ser representado em um grafo correspondente. Um tipo comum de grafo suportado pela maioria dos sistemas é o grafo de propriedade (MILLER, 2013).

A figura 2.11 mostra o comportamento de um grafo de propriedade. O grafo grava nós e relacionamentos. Os nós possuem propriedades, assim como as relações, porém as relações também fazem a organização dos nós.

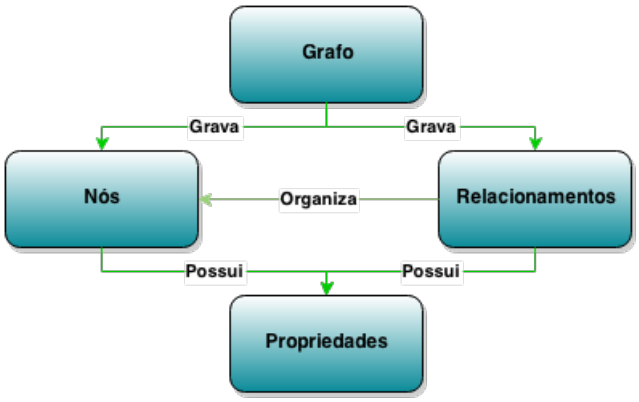

Figura 2.11: Modelo de Dados Orientado a Grafos (NEO4J, 2014)

2.8.4 Comparação dos Modelos de Dados

Os modelos de dados podem ser classificados quanto a performance, escalabilidade, flexibilidade, complexidade e funcionalidade. Fazendo uma síntese de acordo com as pesquisas apresentadas na seção 2.8.3, pode-se comparar os modelos de dados. O modelo chave-valor é de alta performance, escalabilidade e flexibilidade, porém não apresenta nenhuma complexidade de dados e nenhuma funcionalidade. O modelo colunar é de alta performance, escalabilidade e moderada flexibilidade, porém apresenta baixa complexidade de dados e uma funcionalidade mínima. O modelo documental é de alta performance, escalabilidade e flexibilidade, porém apresenta complexidade de dados moderada e uma funcionalidade relativamente baixa. O modelo de grafos pode variar na performance e escalabilidade, porém possui alta flexibilidade e complexidade, além de utilizar a teoria dos grafos como base para funcionalidades.

Por último, o modelo relacional varia em performance e escalabilidade e possui flexibilidade baixa, porém apresenta alta complexidade de dados e utiliza a álgebra relacional como base para funcionalidades. A tabela 2.2 mostra esta comparação dos modelos de dados.

Modelo	Performance	Escalabilidade	Flexibilidade	Complexidade	Funcionalidade
Chave-Valor	alta	alta	alta	nenhuma	variável (nenhuma)
Colunar	alta	alta	moderada	baixa	mínima
Documental	alta	variável (alta)	alta	moderada	variável (baixa)
Grafo	variável	variável	alta	alta	teoria dos grafos
Relacional	variável	variável	baixa	alta	álgebra relacional

Tabela 2.2: Comparação dos Modelos de Dados NoSQL

Uma outra análise pode ser feita a partir da relação do tamanho da base de dados suportada em cada modelo. Percebe-se que quanto maior o tamanho da base, menor é a complexidade do modelo de dados. A figura 2.12 mostra a relação complexidade-tamanho nos modelos de dados. Note que para 90% dos casos de uso, o modelo orientado a grafos e o relacional apresentam uma alta complexidade, porém uma menor capacidade de armazenamento de dados, enquanto que os modelos documental, colunar e chave-valor, exclusivamente nessa ordem, perdem gradativamente a complexidade em benefício de uma maior capacidade de armazenamento de dados.

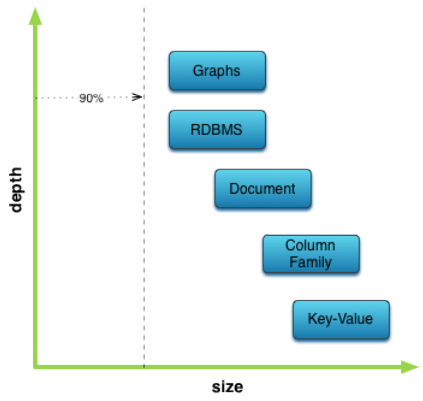

Figura 2.12: Complexidade dos Modelos de Dados NoSQL (NEO4J, 2014)

Este trabalho concentra-se em um modelo de dados com uma complexidade moderada que pode prover funcionalidades relativamente altas. O modelo deve ser de alta performance e escalabilidade, podendo oferecer uma flexibilidade moderada. Com estas características vemos que o modelo chave-valor apesar da performance não atende a complexidade e o modelo orientado a grafos é complexo demais. O modelo colunar é um bom candidato, porém pode exigir uma implementação mais aprofundada de sistemas de arquivos, o que foge do escopo do trabalho.

Já o modelo documental se encontra mais dentro desse propósito, pois sua complexidade moderada talvez possa ser estendida com o modelo relacional, podendo aumentar seu nível de funcionalidades. Logo este modelo foi eleito para este trabalho e com ele foram implementadas funcionalidades e algumas operações para o uso da interface.

2.9 Linguagem de Programação *Python*

A linguagem de programação *Python* é de uso geral e combina funcionalidades da programação estruturada junto com os paradigmas da programação orientada a objetos, geralmente apontada como uma linguagem de *script* com orientação a objetos. De acordo com (LUTZ, 2013), alguns fatores que levam a sua utilização são:

- *qualidade*: legibilidade, coerência, reutilização e manutenção são palavras-chave na sua filosofia de desenvolvimento, além de prover uniformidade e modulação no código, tornando-se fácil de entender, para quem não não está habituado a linguagem.
- *produtividade*: os códigos são tipicamente um terço a um quinto do tamanho dos códigos produzidos em linguagens tipificadas como C, C++ e Java, além dos programas poderem ser executados imediatamente, sem a necessidade de etapas de compilação exigidos por algumas linguagens.
- *portabilidade*: É comum a execução de programas sem a necessidade de alteração nas principais plataformas de computador. Além disso, *Python* oferece várias opções para a codificação de interfaces gráficas para usuário, programas para acesso a dados e sistemas web.
- *biblioteca padrão*: a *standard library* é nativa e fornece um grande número de funcionalidades, podendo ser estendida com bibliotecas próprias e com a vasta quantidade de bibliotecas de terceiros existentes na comunidade.

Neste projeto, a a utilização da linguagem *Python*, além destes motivos, foi muito influenciada por trabalhar muito bem com o formato JSON, pois apresenta uma estrutura de dados diretamente compatível com o formato. Também foram utilizadas bibliotecas de terceiros, para o desenvolvimento da aplicação Web foi utilizado o *framework Pyramid*, para desenvolvimento da interface foi utilizada a biblioteca *pyramid-restler* e para comunicação com o banco foram utilizadas as bibliotecas *psycopg2* e *SQLAlchemy*.

2.10 Banco Relacional

Até agora foram apresentados os SGBDR e NoSQL como duas tecnologias diferentes por este trabalho. Porém o objetivo é promover a coexistência das identidades de cada modelo, partindo de que os dois podem não ser uma oposição um ao outro. Ambos podem compartilhar ideias em comum, um exemplo disso é a estrutura de índices de Árvores-B ou derivadas usada pelos produtos de SGBDR e NoSQL.

Com o modelo de dados documental elegido (seção 2.8.4), pode-se olhar para os produtos existentes e avaliar qual pode encaixar melhor este modelo. Dentre os SGBDR mais populares do mercado estão:

- *Oracle Database* (ORACLE, 2014),
- *Microsoft SQL Server* (MICROSOFT, 2014b),
- *MySQL* (MYSQL, 2014),
- *IBM DB2* (IBM, 2014) e
- *PostgreSQL* (POSTGRESQL, 2014).

Alguns dos produtores destes bancos de dados preferiram desenvolver outras versões NoSQL dos seus produtos com características mais específicas e suporte ao ambiente de nuvem como no caso da *Oracle* (JOSHI; HARADHVALA; LAMB, 2012), e o *Microsoft Azure* (WILDER, 2012), que internamente usa o *SQL Server*.

Os bancos de dados *PostgreSQL* e *IBM DB2* nas suas mais novas versões tem suporte a persistência de objetos JSON, um dos preferidos nesta era de mídia social, computação móvel e nuvem. Há também pesquisas (WHITTAKER, 2013) que relatam uma melhora na performance do *PostgreSQL* usando BSON (Serialização binária de documentos JSON).

Ao *MySQL* pode ser acoplado o *plugin* (programa de computador usado para adicionar funções a outros programas, provendo alguma funcionalidade específica) *HandlerSocket*, que basicamente permite a camada SQL o acesso ao motor de armazenamento e fornece uma interface NoSQL para acesso rápido, especialmente para buscas de chave-primária.

O *PostgreSQL* é um sistema de banco de dados relacional de código aberto. Ele executa a maioria dos padrões *International Organization for Standardization*(ISO) e *American National Standards Institute*(ANSI) mais recentes para a consulta de banco de dados na linguagem SQL, tem suporte a transações ACID, e é totalmente transacional. Dentre as vantagens do PostgreSQL estão a sua disponibilidade para diversas plataformas, arquitetura altamente extensível, suporte para tipos de dados complexos, e maturidade.

Porque é um sistema de banco de dados eficiente e confiável, e porque já tem suporte para o armazenamento de documentos, para este projeto foi decidido trabalhar com o *PostgreSQL* para realizar a persistência e armazenamento dos dados.

Capítulo 3

Modelo de solução proposto

3.1 Etapas de Desenvolvimento

A figura 3.1 mostra o diagrama de etapas para o desenvolvimento do projeto. Nela podem ser observadas as seis principais etapas. Na primeira, foi escolhido o modelo de dados. Como já descrito neste trabalho na seção 2.8.4, o modelo de dados documental foi o escolhido. Na segunda etapa, foi implementado o modelo de entidades (seção 3.7). Na terceira etapa, foi implementada a estrutura do modelo de dados (seção 3.4). Na quarta etapa, foi implementada uma API REST (seção 3.5). Na quinta etapa, foi integrada a API com banco relacional. Nesta etapa e na seguinte, a prática consiste na instalação e configuração de ferramentas e da rede, portanto não é necessário explicitar estas etapas neste trabalho. Na sexta e última etapa, foi configurada a nuvem privada para executar os testes.

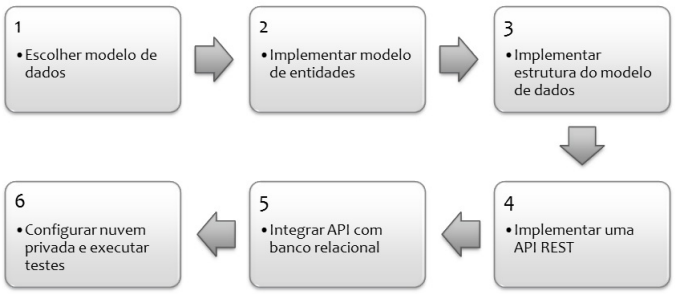

Figura 3.1: Etapas de Desenvolvimento

3.2 Estrutura Geral do Modelo Proposto

A estrutura geral do projeto é dividida em três camadas e engloba o ambiente de comunicação em nuvem privada (seção 2.3), o *software* da interface (seção 2.5), e o banco relacional (seção 2.10), como mostra a figura 3.2. Além destas camadas, a figura mostra

também as aplicações-cliente da ferramenta, fazendo várias requisições, que passam pela nuvem privada até chegar na interface.

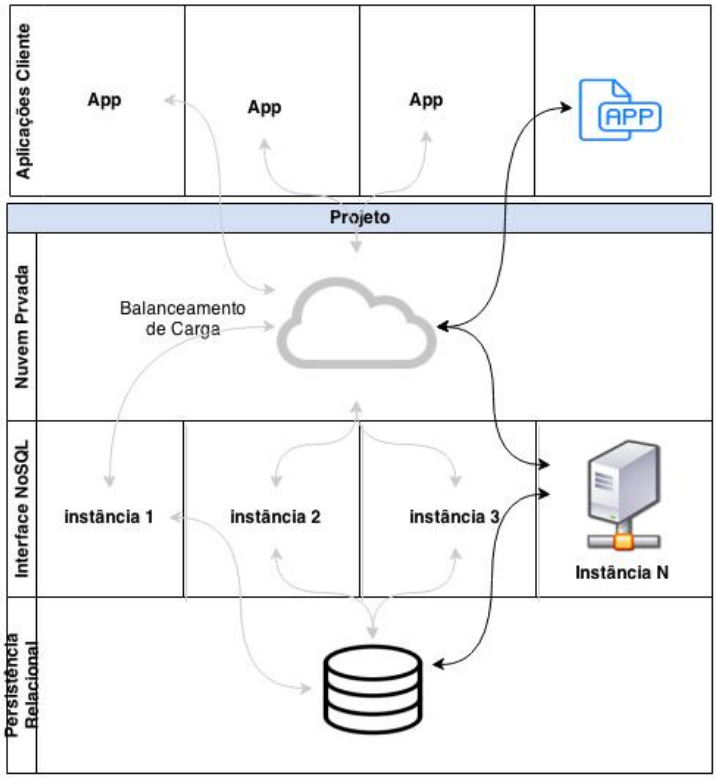

Figura 3.2: Estrutura Geral do Projeto

O ambiente de comunicação em nuvem privada é responsável por realizar a transmissão de informações entre o cliente (aplicações), a interface e o banco de dados, além de prover escalabilidade de processamento. A escalabilidade pode ser feita através da implantação de várias instâncias da interface no ambiente de nuvem privada, com o auxílio de um balanceador de carga para distribuir as requisições.

O software da interface que foi desenvolvido como parte da proposta de projeto aqui apresentada, realiza processamento de requisições e tratamento de dados, além de fornecer métodos através da sua API que possibilitam o gerenciamento de dados.

O banco de dados persiste os dados tratados pela interface. O modelo de tabelas do banco de dados foi implementado neste projeto de forma simples, com poucas tabelas e nenhum relacionamento, como em um banco de dados não relacional.

3.3 Separação Semântica dos Dados

Nos bancos de dados documentais as chaves são mapeadas para documentos que contém informação estruturada. Alguns trabalham com coleções, para que as chaves de departamento ou empregados não se colidam, por exemplo (MARCUS, 2011).

Logo, a separação semântica como nos bancos documentais, separa os conjunto de dados para evitar a colisão de chaves. De forma análoga, é como as tabelas do modelo relacional. Por isso foi implementada uma solução de forma a utilizar o modo de compartilhamento por tabela, com isolamento por linhas, utilizada nos modelos de SaaS como mostra a tabela 2.1 apresentada no capítulo 2. É importante ressaltar que apesar de não fazer parte do projeto o controle de usuários, com este modo de compartilhamento é possível relacionar cada usuário com uma base na tabela de bases.

Por convenção, este trabalho a partir desta página referenciará uma coleção de dados como uma *base* de dados. Portanto, no contexto do trabalho uma base de dados tem um objetivo semelhante à uma coleção dos bancos de dados documentais. É um conjunto definido de dados. A base está no nível máximo de hierarquia existente na interface, todas as operações adjacentes estão relacionadas com uma ou mais bases.

A pesquisa sobre bancos de dados documentais realizada neste trabalho revelou que um esquema dinâmico nem sempre é vantajoso para certas aplicações. Por este motivo, ao projeto da interface interessa fornecer ao desenvolvedor um meio de esquematizar a estrutura dos documentos. Em súmula, a base de dados da interface foi criada neste trabalho para atingir dois objetivos: a separação semântica dos dados e a estruturação de documentos com base em um esquema.

3.4 Esquematização Estrutural dos Dados

Para poder esquematizar os documentos das bases, foi desenvolvido um modelo estrutural (estrutura da base de dados) que pode representar um modelo estrutural dos documentos. A figura 3.3 demonstra o relacionamento dos modelos que compõem o modelo estrutural do esquema das bases.

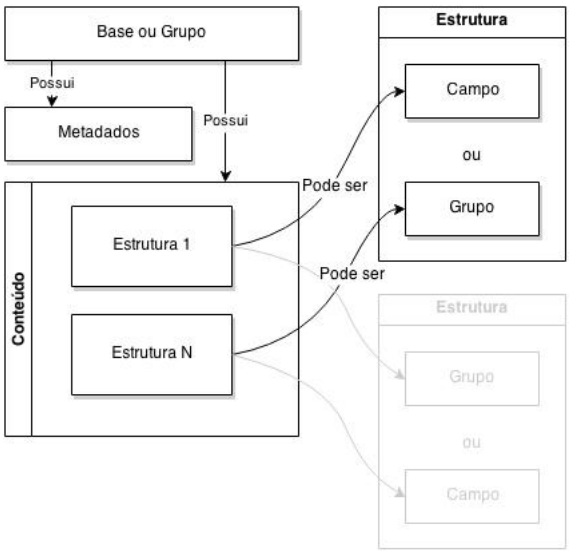

Figura 3.3: Modelo Estrutural do Esquema das Bases

A estrutura da base também é um documento que contém metadados e conteúdo. Os metadados da base contém dados relativos à própria base como nome e identificador. O conteúdo da base é uma lista de todas as estruturas que os documentos podem conter.

As estruturas desenvolvidas são de dois tipos: campos e grupos. Um campo é um espaço para preenchimento de dados definido pelo usuário. Possui atributos como nome, descrição, tipo, obrigatoriedade, índices e multi valoração. Um grupo é um conjunto de campos. Também é composto por metadados e conteúdo, como a estrutura da base. Possui atributos como nome, descrição, e multi valoração.

Os atributos das estruturas são características particulares de cada estrutura definida. O atributo tipo da estrutura, por exemplo, está relacionado com a forma do dado, enquanto que a obrigatoriedade é relativa a possibilidade de preenchimento do dado. Os índices são alguns dos tipos de índices do banco relacional e a multi valoração indica a possibilidade de múltiplos valores (dados) em uma mesma estrutura.

O modelo estrutural da base foi desenhado de forma que seja possível representar um grande número de possibilidades estruturais de um documento. Por isso é uma estrutura recursiva. Isso significa que partes da estrutura podem se repetir dentro delas mesmas. Isso acontece com os grupos, por exemplo. Como cada grupo possui metadados e conteúdo, e o conteúdo é uma lista que pode conter outros grupos, cada grupo pode conter outros grupos. Com este modelo é possível representar objetos aninhados em um documento. A figura 3.4 mostra o diagrama de classes desenvolvidas como parte deste projeto para implementar o modelo estrutural das bases de dados.

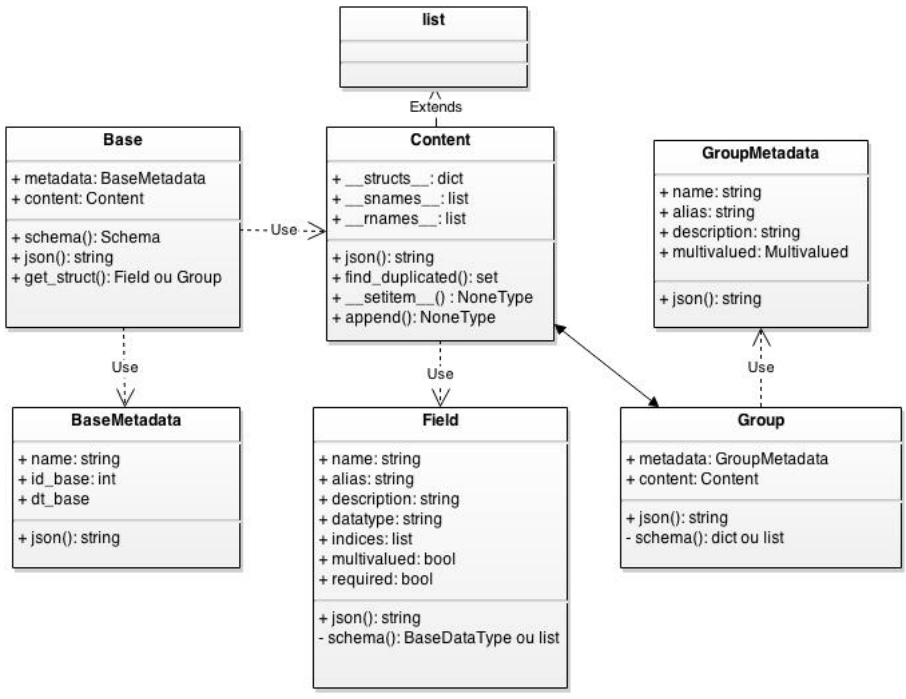

Figura 3.4: Diagrama de Classes do Modelo Estrutural das Bases de Dados

São a seis classes fundamentais da interface. A classe principal (Base) é a classe que

representa o modelo estrutural do documento. O método `json()` foi implementado para transformar o objeto para sua representação JSON, e está presente em todas as seis classes pois cada uma tem sua representação. Este método chama os outros métodos das classes adjacentes para poder construir a representação JSON da base.

Os atributos presentes em cada classe são exatamente os mesmos usados para construir a representação da JSON da base. Com exceção apenas da classe `Content`, que não possui atributos na sua representação, já que é um array. Esta classe possui o método `find_duplicated()` que serve para encontrar nomes de estruturas duplicadas no nível em questão e nos níveis mais altos. A interface foi implementada de forma que não permitirá bases com nomes de estruturas repetidas, afim de facilitar o desenvolvimento e melhorar a performance em alguns casos.

3.4.1 Modelo Estrutural dos Documentos

Para ficar mais claro qual modelo estrutural de documentos é suportado pela interface, é necessário definir cada tipo de estrutura e seus possíveis modelos. A tabela 3.1 mostra os possíveis modelos de cada tipo de estrutura suportada pela base para criação de documentos.

Documento	*{ Membros }*
Membros	*Estrutura* *Estrutura, Membros*
Estrutura	*Campo* *Campo Multi-Valorado* *Grupo* *Grupo Multi-Valorado*
Campo	*string: valor*
Campo Multi-Valorado	*string: []* *string: [Elementos]*
Elementos	*valor* *valor, Elementos*
Grupo	*string: Documento*
Grupo Multi-Valorado	*string: []* *string: [Documentos]*
Documentos	*Documento* *Documento, Documentos*

Tabela 3.1: Modelo de Documento Suportado pela Interface

É importante ressaltar que o campo *string* citado na tabela obedece a mesma notação definida pela notação JSON, mas com algumas exceções. Este nome só aceita caracteres em caixa baixa não acentuados e o caractere sublinhado.

As estruturas de campo e campo multi-valorado são as últimas instâncias de um documento, por isso é onde se localiza o dado real do usuário, representado na tabela por *valor*.

O *valor* dos campos obedecem aos tipos definidos na base, onde apenas as estruturas de campo possuem o atributo tipo. Dentre alguns tipos nativos da interface estão o tipos *Text*, *Boolean*, *Integer* e *File*. O tipo *File* por ser mais complexo, é mais detalhado na seção 3.4.3.

Além das estruturas definidas até agora, existe também uma estrutura especial `_metadata` que guarda alguns dados (ex.: identificador, data de criação e data de alteração) referentes ao próprio documento, ou seja, são os meta dados do documento. A listagem 3.1 mostra um exemplo do campo especial `_metadata`, que representa os metadados do documento.

```
{
    _metadata": {
        "id_doc": 3,
        "dt_doc": "05/08/2014 10:45:35",
        "dt_last_up": "05/08/2014 12:50:22"
    }
}
```

Listing 3.1: Exemplo de meta dados do documento.

Alguns bancos de dados documentais já utilizam esta abordagem, mas geralmente os meta dados estão no primeiro nível na estrutura do objeto. Isso é interessante para o usuário não precisar fazer outras requisições para acessar dados básicos sobre o documento, e além disso, este dados são gravados no banco juntos com o documento, evitando a necessidade de processamento para colocar estes dados virtualmente para devolução ao usuário.

3.4.2 Validação Estrutural dos Documentos

Neste trabalho, a biblioteca de terceiro *voluptuous* foi utilizada para validar a estrutura dos documentos. As classes `Base`, `Group` e `Field` possuem o método `schema()`, que ao ser chamado devolve o esquema relativo à estrutura correspondente. O trecho de código 3.2 mostra o algoritmo responsável por criar o esquema da base.

```
def schema(self, id):

    # Cria dict vazio
    schema = dict()

    for struct in self.content: # Para cada estrutura:

        if struct.is_field: # Caso seja um campo:
            structname = struct.name # Pega o nome
        elif struct.is_group: # Caso seja um Grupo:
            structname = struct.metadata.name # Pega o nome

        if getattr(struct, 'required', False):
            # Se for obrigatorio, construir chave especial
```

```
15              structname = voluptuous.Required(structname)
16
17          # Atualizar dict com os esquemas das sub-estruturas
18          schema.update({structname: struct.schema(self, id)})
19
20      # Devolver esquema pronto
21      return voluptuous.Schema(schema)
```

Listing 3.2: Algoritmo de Criação do Esquema da Base.

Este método navega nas estruturas do conteúdo da base, populando o dicionário `schema` com chaves (nomes das estruturas) e valores (esquemas das sub-estruturas). No final do processamento, é criado um objeto de esquema da base, que é utilizado para validação dos documentos relativos aquela base.

Os esquemas das sub-estruturas que podem existir são os esquemas dos grupos e dos campos. O trecho de código 3.3 mostra o algoritmo responsável pela criação do esquema das estruturas de grupo.

```
1  def schema(self, base, id):
2
3      # Cria dict vazio
4      schema = dict()
5
6      for struct in self.content: # Para cada estrutura neste grupo:
7
8          if struct.is_field: # Caso seja um campo:
9              structname = struct.name # Pega o nome
10         elif struct.is_group: # Caso seja um grupo:
11             structname = struct.metadata.name # Pega o nome
12
13         if getattr(struct, 'required', False):
14             # Se for obrigatorio, construir chave especial
15             # Construir tambem o esquema da sub-estrutura
16             schema[voluptuous.Required(structname)] = struct\
17                 .schema(base, id)
18         else:
19             # Caso contrario, apenas construir o esquema da sub-estrutura
20             schema[structname] = struct.schema(base, id)
21
22     if self.metadata.multivalued is True:
23         # Se este grupo e multivalorado, devolve lista
24         return [schema]
25     elif self.metadata.multivalued is False:
26         # Caso contrario devolve o esquema
27         return schema
```

Listing 3.3: Algoritmo de Criação do Esquema do Grupo.

O método acima é similar ao de criação do esquema da base, com a diferença que o grupo em questão pode ser multi-valorado ou não. Então no final do processamento é feita uma verificação, e caso seja multi-valorado é devolvida uma lista com um elemento, ou caso contrário, é devolvido apenas o esquema do próprio grupo.

Ainda, é possível que os esquemas das sub-estruturas de campos sejam criados. O trecho de código 3.4 mostra o algoritmo responsável pela criação do esquema das estruturas de campo.

```
def schema(self, base, id=None):

    # Pegar o tipo desta estrutura
    datatype = self._datatype.__schema__

    if self.multivalued is True:
        # Caso esta estrutura seja multi-valorada,
        # devolve uma lista
        return [datatype(base, self, id)]

    elif self.multivalued is False:
        # caso contrario devolve o tipo
        return datatype(base, self, id)
```

Listing 3.4: Algoritmo de Criação do Esquema do Campo.

A criação do esquema da estrutura de campo é restrita apenas ao seu tipo. No começo do algoritmo, é guardado o tipo do campo na variável `datatype`, e então a mesma verificação de estrutura multi-valorada é feita. Caso o campo seja multi-valorado, é devolvida uma lista com um elemento, ou caso contrário, é devolvido apenas o tipo do próprio campo.

Os trechos de códigos apresentados mostram apenas a criação do esquema de cada estrutura correspondente. O trecho de código 3.5 mostra o algoritmo responsável pela validação dos documentos de cada base.

```
def validate(self, document, metadata):

    id = metadata.id_doc # Pega o identificador do documento

    # Reserva espaco para os arquivos
    self.__files__[id] = [ ]
    # Reserva espaco para os dados das colunas relacionais
    self.__reldata__[id] = { }

    if '_metadata' in document:
        # Caso o documento possua a chave '_metadata', ignore-a
        del document['_metadata']

    _schema = self.schema(id) # Construir esquema da Base

    try:
        # Tenta validar documento com base no esquema
        document = _schema(document)
    except Exception as e:
        # No caso de erro, limpar espacos de memoria
        del self.__files__[id]
        del self.__reldata__[id]
        # Levantar excecao
        raise exc.ValidationError(e)

    # Colocar de volta os metadados do documento
```

```
27      document['_metadata'] = metadata.__dict__
28
29      # Devolve documento, arquivos e dados das colunas relacionais
30      return (document,
31              self.__reldata__[id],
32              self.__files__[id])
```

Listing 3.5: Algoritmo de Validação dos Documentos.

A função `validate` recebe os parâmetros `document` e `metadata`. Nas primeiras linhas a função guarda o identificador do documento e cria áreas de memória para os arquivos e para os dados que serão guardados em colunas relacionais nos atributos `__files__` e `__reldata__`, respectivamente. logo depois, a função remove a chave referente aos meta dados do documento, e cria o esquema da base chamando o método `schema()`. No bloco de tratamento de exceções é validado o documento, e caso hajam erros, a função limpa os espaços de memória alocados e levanta uma exceção do tipo `ValidationError`. Nas linhas finais, os metadados são colocados de volta no documento e é devolvido o documento, arquivos e dados das colunas relacionais referentes ao documento.

No final do processamento e caso não haja exceções, é criado um objeto documento. Isso quer dizer que o documento é válido e pode ser usado pela interface. Esse algoritmo é usado para inserção e alteração de documentos na interface. Qualquer documento que tenha uma estrutura compatível com a suportada pela base e tem os tipos dos valores de acordo com os tipos definidos na base pode passar por ele sem provocar exceções.

3.4.3 Tipos de Dados em Campos

Os tipos dos campos são definidos na estrutura da base pelo atributo *datatype*. Dentre os tipos suportados pela inteface estão os tipos *Boolean*, *Date*, *File*, *Integer*, *Json* e *Text*.

Alguns dos tipos implementados são nativos da linguagem *python*, e alguns são mais customizados como o tipo *File*. Neste trabalho não detalha cada tipo suportado pela interface, porém esse tipo é interessante pois a arquitetura é um pouco diferente dos outros.

Com o tipo Arquivo (*File*) é possível operacionalizar bytes de arquivos genéricos pela interface. Isso significa que os arquivos podem estar vinculados aos documentos. A engenharia de criação de arquivos foi desenhada afim de que o usuário possa enviar vários arquivos de forma assíncrona, e depois associá-los à um documento. Além do binário do arquivo, o documento também possui seus metadados, como nome, tipo e tamanho em bytes. O valor de um campo do tipo arquivo em um documento é chamado de máscara de arquivo. A listagem 3.6 mostra um exemplo de máscara de arquivo.

```
1 {
2     "mimetype": "text/xml",
3     "filesize": 1321,
4     "id_file": "8fc15324-9015-303e-bb6a-45362d46e346",
5     "filename": "mascara-arquivo.xml"
6 }
```

Listing 3.6: Exemplo de máscara de arquivo.

Esta máscara é retornada por uma requisição HTTP, que continha os bytes de um arquivo no corpo. Ela deve ser inserida no documento no campo desejado. Internamente, o processo de existência de arquivo na interface se divide em duas etapas: a criação e o relacionamento do arquivo com um documento. Na primeira etapa, a interface cria a máscara e salva os bytes e metadados no banco de dados, sem nenhum tipo de ligação com nenhum documento. A figura 3.5 mostra o diagrama de processo de criação de arquivos na interface.

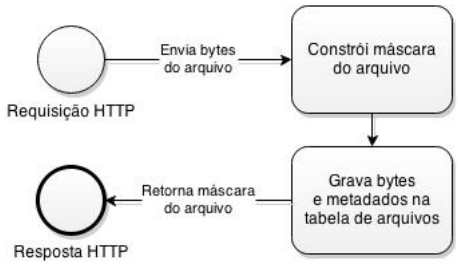

Figura 3.5: Etapa de Criação de Arquivos

Perceba que este processo pode ser feito várias vezes de forma assíncrona. Na segunda etapa, o documento com a máscara é enviado em outra requisição, a interface no processo de validação identifica as máscaras de arquivo presentes no documento, remove os arquivos que existem no banco mas não existem no documento, e por último altera o identificador do documento nos arquivos identificados que ainda não foram ligados à um documento. A figura 3.6 mostra o diagrama de processo de relacionamento de um arquivo com documento na interface.

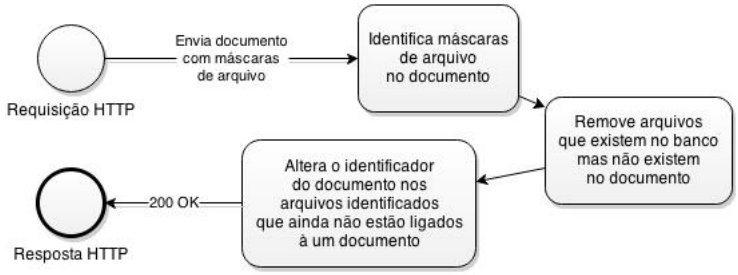

Figura 3.6: Etapa de Relacionamento dos Arquivos com um Documento

Esta arquitetura permite relacionar arquivos em qualquer nível do documento, pois é de responsabilidade do usuário inserir a máscara no documento. A desvantagem é que alguns arquivos criados na primeira etapa não são relacionados com um documento, caso o usuário não o faça por qualquer motivo gerando assim uma carga de dados inútil para o banco. Uma possível solução para este problema seria implementar um coletor de lixo que removesse do banco os arquivos criados à mais de uma hora, por exemplo.

3.5 A Interface REST

Esta camada é o objeto central do projeto, pois é a camada que realiza as operações de gerenciamento de dados. Além disso ela fornece a comunicação entre o usuário (aplicações) e o banco de dados.

A operacionalização da interface é dividida em três segmentos: base, coleção e caminho. A base foi apresentada na seção 3.3 deste trabalho. A base é um conjunto definido de dados composta pela da estrutura da base, um documento JSON que representa a definição do modelo de dados dos documentos (objetos) de uma coleção.

Uma coleção é um tipo de entidade operacional. Representa o tipo de operação à ser processada pela interface. Atualmente, a interface trabalha com dois tipos de coleções: documentos e arquivos.

O segmento caminho é representado pelos níveis (nós) da estrutura de árvore do objeto. Um nível pode ser uma chave para um objeto ou um índice inteiro de um array.

Estes três segmentos foram implementados de forma fornecer uma filtragem de recursos na interface. Como descrito na seção 2.4 deste trabalho, a arquitetura REST apresenta uma interface uniforme onde cada recurso é identificado e cada parte pode ser trabalhada independentemente.

Com base nessa arquitetura, cada segmento da operacionalização da interface pode ser trabalhada de forma separada. De forma a seguir as restrições arquitetônicas de APIs para serviços Web, é necessário seguir alguns aspectos:

- Uma URI básica como http://example.com/.
- Um tipo de mídia da Internet para dados. Como já citado, a interface trabalha com JSON.
- Os métodos HTTP padronizados (GET, PUT, POST, DELETE).

A URI pode se estender afim de prover uma interface uniforme e a filtragem de dados. Com o objetivo de utilizar uma extensão da URI foi criada uma estrutura lógica ou assinatura de rotas genérica baseada em hierarquia para a interface. A assinatura pode ser vista da seguinte forma:

<base>/<coleção>/<identificador>/<caminho>

Onde *<base>* é o local definido para preencher a base de dados, *<coleção>* o local definido para preencher o tipo da coleção, *<identificador>* o local definido para preencher o identificador único do objeto e *<caminho>* o local definido para preencher os níveis na estrutura de árvore do objeto.

O identificador é um *token* léxico único usado para identificar (representar) um objeto. O `caminho` segue o mesmo princípio da assinatura genérica, com a diferença de que seu tamanho é variável. É usado para selecionar uma parte específica do objeto. Pode ser representado da seguinte forma:

$$N_1/N_2/N_3/.../N_n$$

Em que cada N representa um nó da estrutura de ávore do objeto. A idéia básica é que cada parte da assinatura possa fornecer as quatro operações fundamentais do REST (get, post, put, delete). Como tem um comportamento hierárquico, uma coleção sempre está ligada à uma base, um identificador sempre está ligado à uma coleção, e um caminho sempre está ligado à um objeto.

De maneira geral, a interface deve receber uma requisição HTTP, verificar o método da requisição, encontrar a rota correspondente, executar o método da API, gerar o resultado e responder pelo mesmo protocolo. Caso a rota não seja encontrada, levanta-se uma exceção, gerando outro resultado e depois gera a resposta HTTP. A figura 3.7 mostra o diagrama de processo interno da interface.

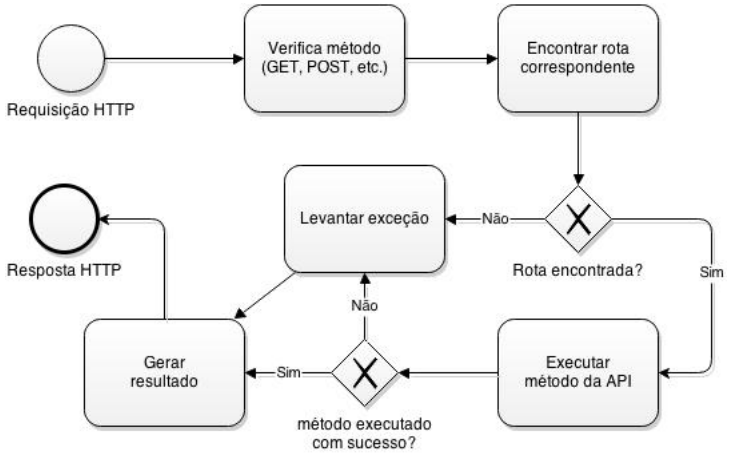

Figura 3.7: Processo Interno da Interface REST

Para poder operar na Web, foi utilizado o *Pyramid* (PYRAMID, 2014), um *framework* de código aberto para desenvolvimento de aplicações Web.

O *Pyramid* permite de forma simplificada adicionar rotas e associá-las a métodos de classes customizadas. No projeto, as classes ficaram divididas em duas camadas. Na primeira camada o métodos geralmente são usados para gerar o resultado de uma requisição HTTP, ou seja uma resposta HTTP. Os métodos então manejam os cabeçalhos, e o corpo da página HTTP. Junto com as classes da primeira camada estão atreladas outros métodos das classes de contexto, que já fazem parte da segunda camada e geralmente são usadas para gerenciar a sessão com o banco de dados, além de realizar a comunicação direta com o mesmo. A figura 3.8 mostra o diagrama de classes desta primeira camada da interface.

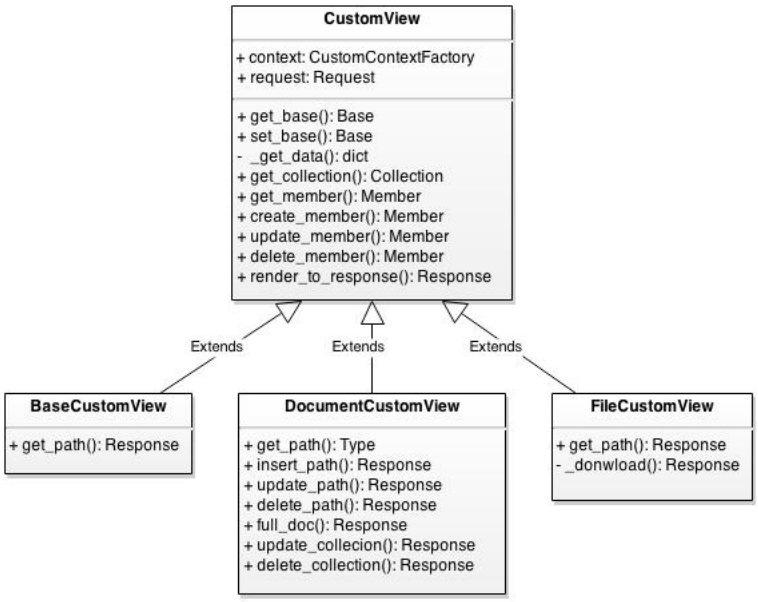

Figura 3.8: Diagrama de Classes da Interface REST

A classe `CustomView` é uma abstração das classes que a herdam. O atributo `context` refere-se à classe da segunda camada. O atributo `request` é um objeto do tipo `Request`, uma classe do *Pyramid* que contém todos os atributos referentes à uma requisição HTTP. O restante dos métodos desta classe estão mapeados à cada rota definida nas configurações da interface. As classes `BasecustomView`, `DocumentCustomView` e `FileCustomView` são herdadas da classe `CustomView` para possuir os mesmos métodos e atributos, mas a estas são implementados alguns métodos peculiares de cada entidade. Quando uma rota requisitada precisa dos dados do banco, os métodos da classe de contexto são invocados. A figura 3.9 mostra o diagrama de classes da segunda camada da interface.

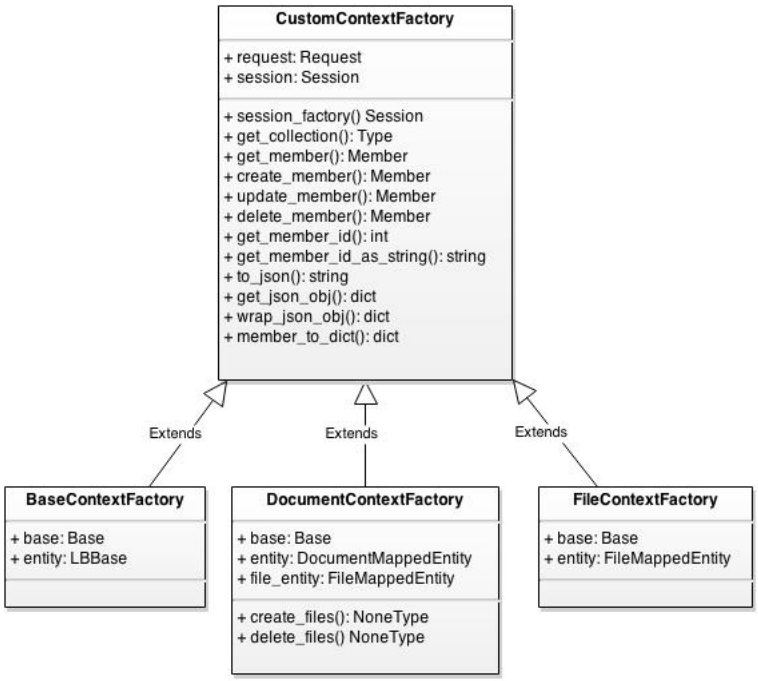

Figura 3.9: Diagrama de Classes de Contexto da Interface REST

Não há muitas mudanças estruturais na modelagem dessas classes, porém o propósito difere no que se trata da comunicação com o banco. A classe principal `CustomContextFactory` possui o atributo `session`, o qual trata do controle de sessão do banco e o atributo `request`, já visto anteriormente. Os métodos desta classe são na sua maioria os mesmos presentes na primeira camada da interface, mas com o propósito geral de efetuar o controle dos dados.

3.6 Métodos da API

3.6.1 Operações com Bases

Antes de começar a citar as operações, é importante informar que cada método HTTP tem uma finalidade. O método POST serve para inserção de dados, o GET para recuperação, o PUT para alteração e o DELETE para remoção. Sendo assim, as operações com bases se dão da seguinte forma (considere como exemplo uma base de nome `users`):

`POST /`	Cria base
`GET /users`	Recupera a estrutura da base
`PUT /users`	Altera a estrutura da base
`DELETE /users`	Remove a base

Nos casos de criação e alteração de bases, o corpo da requisição HTTP deve conter a nova estrutura da base correspondente.

3.6.2 Coleções

A interface suporta dois tipos de coleções: documentos e arquivos. Na assinatura de rotas a representação para documentos é `doc`, e para arquivos é `file`. Nos tópicos seguintes estão descritos com mais detalhes cada uma delas.

3.6.3 Operações com Documentos

Os documentos são um tipo de coleção que a interface possui. A interface foi construída para na inserção e alteração aceitar apenas documentos cuja estrutura esteja de acordo com a definida no modelo de documentos pela estrutura da base. A tabela 3.1 mostra o modelo de documento genérico que uma base pode definir.

Os modelos que não estão presentes na tabela são os mesmos definidos pela notação JSON. No caso do *valor* dos campos, a rotina de validação do documento exige que sejam obedecidos os tipos de dados suportados pela base.

As operações com documentos tem comportamento semelhante às bases. A seguir são exemplificadas operações relativas ao documento com identificador `62616269` da base `users`:

`POST /users/doc`	Cria documento
`GET /users/doc/62616269`	Recupera documento
`PUT /users/doc/62616269`	Altera documento
`DELETE /users/doc/62616269`	Deleta documento

Nos casos de criação e alteração de documentos, o corpo da requisição HTTP deve conter o novo documento correspondente. O documento deve obedecer à estrutura definida na base para que a operação seja efetuada com sucesso. Ainda sobre documentos simples, também foram implementadas operações usando o conceito de caminho do objeto documento:

`POST /users/doc/62616269/fones`	Insere valor no nó
`GET /users/doc/62616269/fones/0/cell`	Recupera valor do nó
`PUT /users/doc/62616269/fones/0/home`	Altera valor do nó
`DELETE /users/doc/62616269/fones`	Deleta nó

As operações com caminho do documento servem para gerenciar parte do documento. O nó `fones` no exemplo é um grupo multi-valorado que contém dois campos (`cell` e `home`). Nos casos de alteração de inserção, o corpo da requisição HTTP deve conter o

novo objeto correspondente. É importante destacar que as operações de inserção usando o caminho dos objetos é suportada apenas para estruturas multi-valoradas.

3.6.4 Operações com Coleção de Documentos

Coleções de documentos são afetadas conjuntamente quando oculta-se o identificador do objeto de uma coleção. As operações com coleções são limitadas afim de evitar gargalos de performance e deleção de dados feitas por requisições errôneas. As operações relativas à coleções de documentos da base `users` se dão da seguinte forma:

`GET /users/doc`	Recupera coleção de documentos
`PUT /users/doc`	Altera coleção de documentos
`DELETE /users/doc`	Deleta documento

No segundo exemplo, a alteração de documentos deve ser feita através dos caminhos dos objetos. No corpo da requisição deve estar presente uma lista de caminhos e objetos à serem alterados nos documentos.

3.6.5 Operações com Arquivos

Como a remoção de arquivos é feita removendo a máscara do documento, o método de remoção de arquivos não é implementado. A alteração de arquivos também não foi necessária, uma vez que é se pode criar um novo arquivo e usar a máscara no lugar do antigo. As operações com arquivos então são as seguintes (considere a base `users` e o arquivo com identificador `06091d1d`):

`GET /users/file`	Recupera coleção de arquivos
`POST /users/file`	Cria arquivo
`GET /users/file/06091d1d`	Recupera atributos do arquivo

A resposta para requisição de recuperação dos arquivos é uma lista com objetos similares a máscara de arquivo citada na seção 3.4.3. Seguindo o princípio da recuperação do caminho dos objetos ainda é possível efetuar as seguintes operações:

`GET /users/file/06091d1d/mimetype`	Recupera tipo do arquivo
`GET /users/file/06091d1d/filetext`	Recupera texto do arquivo
`GET /users/file/06091d1d/filesize`	Recupera tamanho em bytes do arquivo
`GET /users/file/06091d1d/download`	Recupera bytes do arquivo
`GET /users/file/06091d1d/filename`	Recupera nome do arquivo
`GET /users/file/06091d1d/id_doc`	Recupera identificador do documento

3.7 Modelo Relacional

O modelo de dados relacional define entidades altamente estruturadas com relações estreitas entre elas. Consultando este modelo com SQL permite busca de dados complexas e sem muita personalização. A complexidade de tal modelagem e consulta tem seus limites, no entanto (MARCUS, 2011):

- Complexidade leva a imprevisibilidade. A expressividade do SQL torna um desafio para raciocinar sobre o custo de cada consulta. Embora as linguagens de consulta simples poderem complicar a lógica da aplicação, tornam mais fácil a resposta para os sistemas de armazenamento de dados.
- Se os dados crescem além da capacidade de um servidor, as tabelas no banco de dados terão de ser divididas entre computadores. Para evitar que JOINs tenham que atravessar a rede, a fim de obter dados em tabelas diferentes, é preciso desnormalizar o banco de dados.

O modelo de tabelas apresentado como parte deste projeto foi pensado de forma em que as consultas devem ser simples porém com certo nível de complexidade. Os documentos serão esquematizados pela estrutura da base, porém com uma certa flexibilidade na estrutura dos dados. As tabelas não terão qualquer chave estrangeira ou qualquer tipo de relacionamento, afim de promover a desnormalização dos dados e consequentemente, facilitar a escalabilidade do banco de dados.

Para resolver a estruturação dos dados foi implementada a tabela `lb_base`, que persiste as bases disponíveis do sistema. A tabela possui a chave primária `id_base`, a coluna `name` para o nome da base e a coluna `struct` para a estrutura da base, apresentada da seção 3.4 deste trabalho. A figura 3.10 mostra o modelo da tabela de bases.

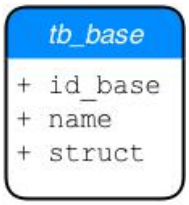

Figura 3.10: Tabela de Bases

Cada registro da tabela de bases possui documentos e arquivos. Em favor da performance, a tabela de documentos é implementada separadamente da tabela de arquivos. Logo, Na criação de uma base, a interface cria então mais duas tabelas, a de documentos e a de arquivos.

Isso significa que para cada registro na tabela de bases, haverão outras duas tabelas no banco de dados. Essas tabelas terão seus nomes baseados no nome da base. Como exemplo, na criação da base `users` serão criadas as tabelas `lb_doc_users` e `lb_file_users`.

Todas as tabelas de documentos tem como prefixo `lb_doc_`, e todas as tabelas de arquivos tem como prefixo `lb_file_`. Se uma base é removida da tabela de bases, automaticamente suas tabelas de documentos e de arquivos são removidas do banco. As tabelas de documentos então são criadas e removidas dinamicamente pela interface. Por este motivos, elas não tem um nome fixo. A figura 3.11 mostra a tabela de documentos.

Figura 3.11: Tabela de Documentos

A coluna `id_doc` é a chave primária da tabela. A coluna `dt_doc` guarda a data de criação do documento, a coluna `dt_last_up` guarda a data da última alteração feita naquele documento, e a coluna `document` guarda o documento em si. Os três pontos (...) mostrados na tabela representam outras colunas definidas dinamicamente pela estrutura da base, onde os campos são definidos com índices do tipo relacional.

O outro modelo de tabela refere-se aos arquivos de cada documento. A figura 3.12 mostra a tabela de arquivos.

Figura 3.12: Tabela de Arquivos

A coluna `id_file` é a chave primária da tabela. A coluna `id_doc` guarda a chave primária da tabela de documentos relacionada, a coluna `filename` guarda o nome do arquivo, a coluna `filesize` guarda o tamanho do arquivo em bytes, a coluna `mimetype` guarda o tipo do arquivo, e a coluna `file` guarda os bytes do arquivo.

Capítulo 4

Aplicação prática da solução proposta

A aplicação prática, envolvendo um caso real de aplicação, tem como objetivo principal mostrar a viabilidade da proposta de resolução sugerida no trabalho, bem como permitir que novos conhecimentos sejam incorporados.

Neste capítulo serão fundamentados, com base na implementação prática, comparações de consultas, simulações e testes de desempenho, mostrando como o trabalho pode ser aplicado em uma aplicação prática.

4.1 Apresentação da Área de Aplicação da Solução

Diferentes aplicações utilizam diferentes modelos de bancos de dados. A ideia geral para solucionar esta problemática é usar a ferramenta certa para a tarefa certa. Isto pode ser exemplificado com uma plataforma de comércio eletrônico, como na figura 4.1.

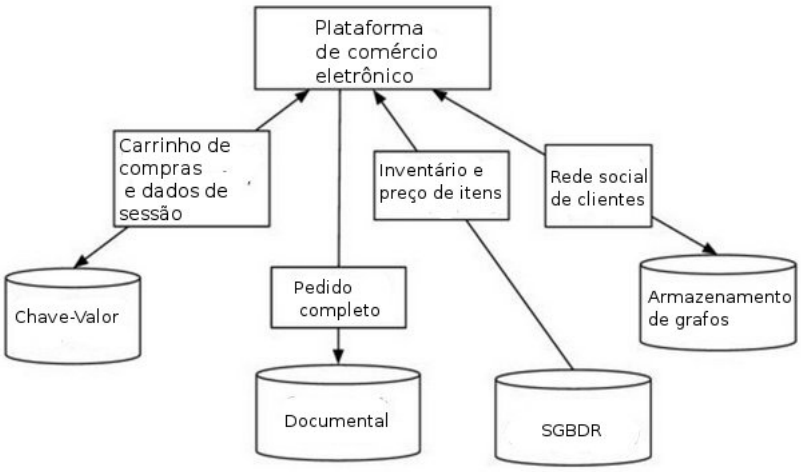

Figura 4.1: Plataforma de comércio eletrônico. Adaptado de (ZERIN, 2013)

A figura mostra que uma mesma aplicação pode ter mais de um modelo para atender bem os requisitos do negócio. No caso do comércio eletrônico, por exemplo, para gerenciar o carrinho de compras e dados de sessão o melhor seria o modelo chave-valor. Já para armazenamento dos pedidos completos um banco documental seria mais adequado, assim como para inventário e preço de itens o modelo relacional se comportaria bem e para a rede social dos clientes caberia o armazenamento de grafos.

Para o modelo apresentado por este trabalho, um bom aproveitamento pode ser feito por aplicações para a web ou para dispositivos móveis, que utilizam bem o modelo documental (FOTACHE; COGEAN, 2013), e que tem uma necessidade a mais, como o suporte à transações ACID por exemplo.

De acordo com (DARROW, 2013), estas aplicações existem e em conjunto com uma solução híbrida, talvez pudessem se comportar melhor. Além disso, o número de aplicações assim como os bancos de dados crescem justamente com o intuito de prover um amplo número de escolhas para o usuário.

4.2 Comparação de Consultas e Comandos

Nos bancos de dados SQL, o esquema é composto por tabelas (*views*, *procedures*), cada tabela tendo uma estrutura comum para todas as suas linhas. Equivalente às tabelas, a interface possui coleções de documentos. No entanto, o modelo de dados é bastante diferente na interface desenvolvida neste projeto, portanto existem algumas opções do SQL sem equivalência para a interface e vice-versa.

Nesta seção é usada a sintaxe SQL do PostgreSQL para uma breve comparação de como são feitas consultas relacionais e não relacionais desenvolvidas como parte deste trabalho.

Os objetos de armazenamento em bancos de dados SQL - tabelas - Podem ser exibidos, consultando o dicionário de dados. No PostgreSQL a sintaxe é:

```
SELECT table_name FROM information_schema.tables
WHERE table_schema = "public" AND table_type
= "BASE TABLE"
ORDER BY table_name;
```

Na interface NoSQL, objeto da solução aqui proposta, os dados são armazenados em bases, as quais podem ser mostradas como abaixo, na consulta com base no modelo da solução proposta:

```
GET /?$$={"select":["name"],"order_by":{"asc":["name"]}}
```

Isso é na verdade uma requisição HTTP usando o método GET na raiz (/) do domínio passando como parâmetro a variável `$$` e um JSON (como valor). Esta sintaxe será usada aqui para representar uma requisição na interface.

Apesar de nos bancos SQL haver uma clara diferença entre DDL (*Data Definition Language*) e DML (Data Manipulation Language), nos bancos NoSQL este conceito não

existe. Não é possível definir os dados como na cláusula SQL `CREATE TABLE`, pois a estrutura dos documentos é criada em tempo de execução. Porém, na interface é possível criar, alterar e deletar bases como mostrado na seção 3.6.1 deste trabalho.

Como apontado anteriormente, outra grande diferença está relacionada com a estrutura de registro. Nas tabelas SQL, cada linha tem uma estrutura (tabular) semelhante. Ao inserir uma linha sem declarar os valores para todos os atributos, os atributos não especificados terão por padrão o valor `NULL`:

```
INSERT INTO estados
VALUES ('DF', 'Distrito Federal', 'Brasilia');
```

Mas na interface, cada documento pode ter um número diferente de atributos (a não ser que todos os campos foram definidos na base como obrigatórios):

```
POST /estados/doc?value={
"uf":"DF", "nome_estado":"Distrito Federal",
"capital":"Brasilia"}
```

Como amplamente reconhecido, os bancos SQL permitem a declaração das seguintes restrições:

- chave primária
- chave única
- restrição *NOT NULL*
- integridade referencial
- regras de validação nos níveis de atributo e registro (*check constraints*)

A interface pode implementar chave primária, única e NOT NULL por meio de índices. Com a estrutura da base também é possível implementar regras de validação limitadas (tipo dos campos) como visto no capítulo 3. O índice de chave única que pode ser criado da seguinte forma no SQL:

```
ALTER TABLE estados ADD UNIQUE (nome_estado);
```

Também pode ser criado de forma equivalente adicionando o índice `Unico` na lista de índices do campo em questão na estrutura da base.

Uma das principais características do SQL é a cláusula `SELECT` usada para expressar consultas com diferentes graus de complexidade. Na interface, foram implementadas técnicas para consulta de banco de dados, algumas delas mostradas abaixo. para consultas básicas, com ordenação e técnicas de ranqueamento, o SQL pode ser escrito da seguinte maneira:

```
SELECT * FROM estados ORDER BY nome_estado DESC
OFFSET 10 LIMIT 20;
```

Na interface NoSQL, objeto da solução aqui proposta, a mesma consulta teria sua equivalência como abaixo:

```
GET /estados/doc?$$={"select":["*"],
"order_by":{"desc":["nome_estado"]},
"offset":10, "limit":20}
```

Tanto no SQL quanto na interface podem aparecer o valor NULL. O usuário pode querer extrair apenas objetos nulos ou que possuam um determinado valor:

```
SELECT capital
FROM estados WHERE nome_estado IS NULL
OR capital = 'Brasilia';
```

Parte desta consulta pode ser reutilizada na interface. Aproveitando da integração com banco relacional, a cláusula `WHERE` foi implementada com o atributo `literal`:

```
GET /estados/doc?$$={"select":["capital"],
"literal":"nome_estado IS NULL OR capital = 'Brasilia'"}
```

Perceba que qualquer cláusula `WHERE` que se encaixe no SQL pode ser usada neste atributo. Assim, o usuário pode executar consultas mais complexas usando a sintaxe da linguagem.

Enquanto que nos bancos NoSQL, a extração de valores distintos podem ser bem complicadas, podendo ser resolvidas com mais de uma consulta, no SQL é bem simples:

```
SELECT DISTINCT capital
FROM estados ORDER BY capital;
```

Esta consulta também simples na interface é feita apenas com a adição do atributo `distinct`:

```
GET /estados/doc?$$={"select":["capital"],
"order_by":{"asc":["capital"]},
"distinct":"capital"}
```

Pode-se citar também a tarefa básica de contar o retorno dos resultados. Com SQL isso pode ser feito com a consulta:

```
SELECT count(*)
FROM estados WHERE nome_estado IS NULL
OR capital = 'Brasilia';
```

A interface oferece como padrão a contagem de resultados para qualquer consulta realizada. Entretanto apenas a contagem pode ser retornada com a lista do atributo `select` vazia:

```
GET /estados/doc?$$={"select":[],
"literal":"nome_estado IS NULL
OR capital = 'Brasilia'"}
```

Para efeitos de comparação, não é necessário mergulhar em detalhes das possíveis consultas, mas desenvolver algumas diferenças em termos de definição, manipulação e consultas de dados que podem ocorrer, mostrando a essência das utilidades desenvolvidas.

Além das citadas acima, pode-se citar também quais são as operações de atualização e consulta de registros nas tabelas podem ser traduzidos para a interface, porém não só na atualização/consulta de uma coleção de documentos, mas também atualização/consulta de subdocumentos ou um *array* em um documento.

Um exemplo simples seria atualizar um valor do atributo em um registro, ou seja, atualização do número de telefone para o cliente 'Cliente 1'. No SQL é possível graças a um comando `UPDATE`:

```
UPDATE clientes SET telefone = '0232217001'
WHERE nome_cliente = 'Cliente 1'
```

Para realizar algo similar na interface, é necessário um segundo parâmetro `path`. Este parâmetro é o caminho para o atributo (campo), com seu respectivo valor. Assim, um comando equivalente seria:

```
PUT /clientes/doc?$$={"literal":"nome_cliente = 'Cliente 1'"}
& path=telefone & value=0232217001
```

Em outros casos, o atributo `path` pode não se restringir à apenas campos no primeiro nível, podendo estender-se à campos dentro de grupos. Neste caso, se `telefone` fosse um grupo que compõe de dois campos (`celular` e `casa`) e fosse necessário atualizar um desses campos, então comando seria um puco diferente:

```
PUT /clientes/doc?$$={"literal":"nome_cliente = 'Cliente 1'"}
& path=telefone/celular & value=0232217001
```

O parâmetro `path` pode ser usado de várias formas, incluindo o caractere *wild-card*(*)(que pode ser substituído por qualquer um de um subconjunto definido de todos os caracteres possíveis) ou até um *slice*, uma notação do *python* que permite selecionar partes de um *array*.

4.3 Metodologia de Testes

Como a interface opera em um ambiente de nuvem, respondendo requisições pela rede, a metodologia de testes que se julgou mais adequada foi a realização de testes de desempenho com requisições HTTP. O teste de desempenho é um tipo de teste destinado a determinar o tempo de resposta, a confiabilidade e a escalabilidade da aplicação sob uma determinada carga de trabalho.

Para realização dos testes de desempenho foi utilizada a ferramenta Jmeter (JMETER, 2014). O aplicativo de *desktop* JMeter é um software de código aberto, uma aplicação Java projetada para testar carga, comportamento funcional e medir desempenho. Foi originalmente projetada para testes de aplicações web, mas desde então, expandiu-se para outras funções de teste.

O Jmeter permite configurar o número de usuários, porém como a demanda de usuários é desconhecida, julgou-se adequado para este trabalho fazer testes gradativos, para ter conhecimento de para qual número de usuários o sistema se comporta com uma baixa taxa de erros.

Para análise de resultados, foram traçados dois tipos de gráficos que a ferramenta fornece, o gráfico de resultados, e o visualizador *Spline*. De acordo com a documentação da ferramenta, O gráfico de resultados gera um gráfico simples que traça todos os tempos da amostragem. Ao longo da parte inferior do gráfico, os seguintes dados são apresentados:

- Número de amostras (preto): numero total de requisições feitas pela ferramenta.
- Desvio padrão (vermelho): tempo em milissegundos dos casos em que determinadas amostras se distanciam do comportamento médio das demais amostras em razão do tempo de resposta. Quanto menor este valor mais consistente é o padrão de tempo das amostras coletadas.
- Última amostra (preto): tempo em milissegundos da resposta da última requisição.
- Taxa de vazão (verde): razão entre o número total de requisições pelo tempo em minutos.
- Média (azul): média em milissegundos do tempo de resposta de todas as amostras.
- Mediana (roxo): valor em milissegundos que divide as amostras em duas partes iguais. Metade das amostras são menores que a média e a outra metade maior que a média, podendo ter algumas amostras com valor igual a média.

Ainda, o visualizador *Spline* fornece uma visão de todos os tempos de amostragem, desde o início do teste até o fim, independentemente de quantas amostras foram tomadas. O gráfico tem 10 pontos, cada um representando 10% das amostras, e conectados usando uma simples lógica para mostrar uma única linha contínua. O gráfico é automaticamente redimensionado para caber dentro da janela.

O ambiente de testes é composto por cinco computadores (C1, P1, S1, S2 e B1) com hardware diferentes, e as seguintes configurações:

- C1, denominado cliente: sistema operacional GNU/Linux Ubuntu 12.04 LTS 64-bit, processador Intel Core i7-2630QM CPU 2.00GHz, 8 GB de memória RAM e disco 80 GB (5400 rpm);
- P1, denominado *proxy*: sistema operacional GNU/Linux Debian 64-bit, processador Intel Celeron E3400 CPU 2.60GHz, 2 GB de memória RAM e disco 500 GB (5400 rpm);
- S1, denominado servidor: sistema operacional Windows 8 64-bit, processador Intel Core i7 3610QM (2.3 GHz), 8 GB de memória RAM e disco 750GB SATA (5400 rpm);
- S2, também servidor: sistema operacional Windows 8.1 64-bit, Intel Core i7 4500U 2.4 GHz, 8 GB de memória RAM e disco 1TB (5400 rpm);

- B1, denominado servidor de banco de dados: sistema operacional GNU/Linux Ubuntu 12.04.2 LTS, processador Intel Core i7-3770 CPU 3.40GHz, 1 GB de memória RAM e disco 50 GB (5400 rpm).

As máquinas virtuais instaladas nas máquinas servidoras S1 e S2 foram configuradas com sistema operacional CentOS 6.5 64-bits, 2 processadores de 2 núcleos cada, 2GB de memória e 20 GB de espaço em disco. A figura 4.2 mostra a topologia de testes realizados neste trabalho, cujas configurações estão identificadas conforme definição do ambiente de teste exposto.

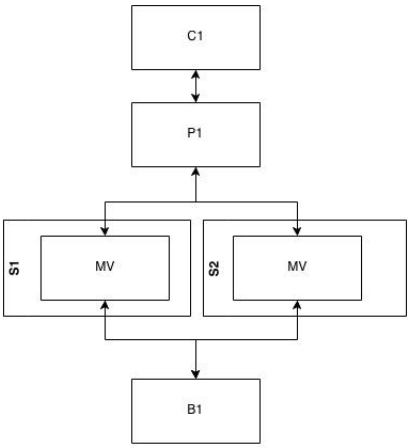

Figura 4.2: Topologia de Testes. Fonte: Autor

A Em uma visão geral, a máquina C1 faz requisições na máquina P1, configurada como balanceador de carga com o *software nginx* (NGINX, 2014) instalado, que distribui as requisições para as máquinas virtuais (MV) hospedadas em S1 e S2, que por sua vez, se comunicam com o servidor postgreSQL na máquina B1.

4.3.1 Cenário de Testes

Quanto ao cenário dos testes de desempenho, outros testes feitos com as ferramentas mais conceituadas do mercado podem focar no tempo de resposta, como na abordagem de (TOTH, 2014), ou na vazão da carga de trabalho como aponta (CASSANDRA, 2014).

Os testes feitos neste trabalho são voltados para o tempo de resposta, e comparados com teste de terceiros, que mostram as ferramentas com latência média abaixo de 100 milissegundos. Portanto, na avaliação global da solução (seção 4.5), são analisados os valores obtidos, e este valor é tido como referência.

4.4 Resultados dos Testes

Nesta seção, são apresentados cenários da implementação do projeto aplicado. São mostrados resultados a partir de gráficos sobre a parte prática específica. Os resultados são apresentados com base no direcionamento para a aplicação proposta.

Como citado na seção anterior, foram feitos testes variando o número de usuários para dar uma ideia da taxa de erro sustentada pelo ambiente de testes. Pode-se considerar como erro uma requisição que não obteve resposta pelo servidor. A figura 4.3 mostra o gráfico de barras da taxa de erro por número de usuários.

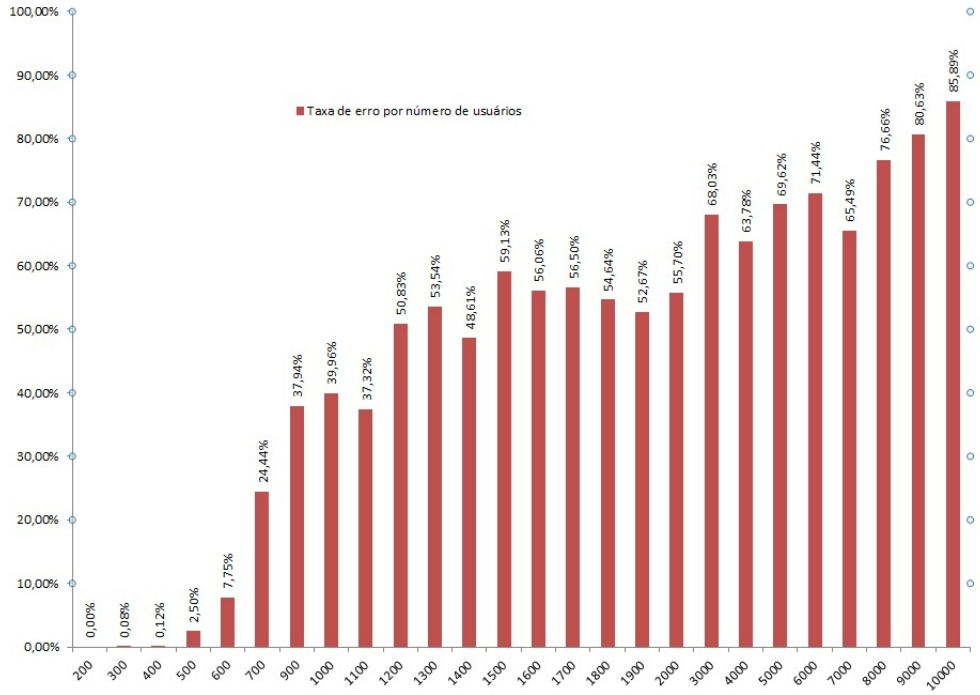

Figura 4.3: Taxa de Erro por Número de Usuários. Fonte: Autor

Como é possível ver no gráfico, o número de usuários que não obteve taxa de erro é menor ou igual a 300 usuários. Portanto, para os testes realizados neste trabalho, considere que este número foi configurado na ferramenta.

Como são muitas as possibilidades de operações, para os testes foram realizadas três operações para dar ideia geral do desempenho da interface, sendo estas de inserção de dados, consulta com limite fixo, e consulta unitária. Cada uma dessas operações é apresentada nas seções seguintes.

4.4.1 Inserção de Dados

No teste de inserção de dados, foi criada uma base com todos os tipos de estruturas possíveis, totalizando 20 atributos no documento. Cada requisição resultará na inserção deste documento no banco de dados pela interface. No corpo de cada requisição enviada pela ferramenta está presente uma amostra desse documento. A figura 4.4 mostra o gráfico de resultados traçado pela ferramenta neste teste.

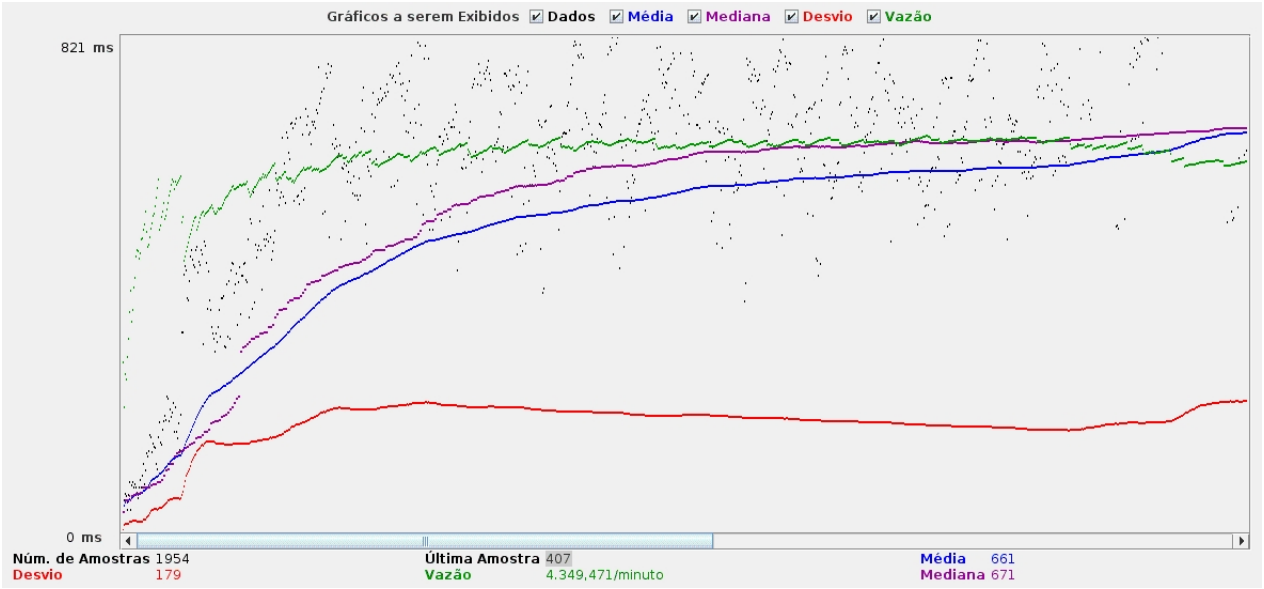

Figura 4.4: Gráfico de Resultados (Inserção de Dados). Fonte: Autor

Como mostra a figura, os dados coletados a partir do gráfico foram de número de amostras (1954), taxa de desvio (179 ms), tempo da última amostra (407 ms), taxa de vazão (4.349.471 req./min), média (661 ms) e mediana (671 ms). A figura 4.5 mostra a visualização *Spline* do mesmo teste de desempenho.

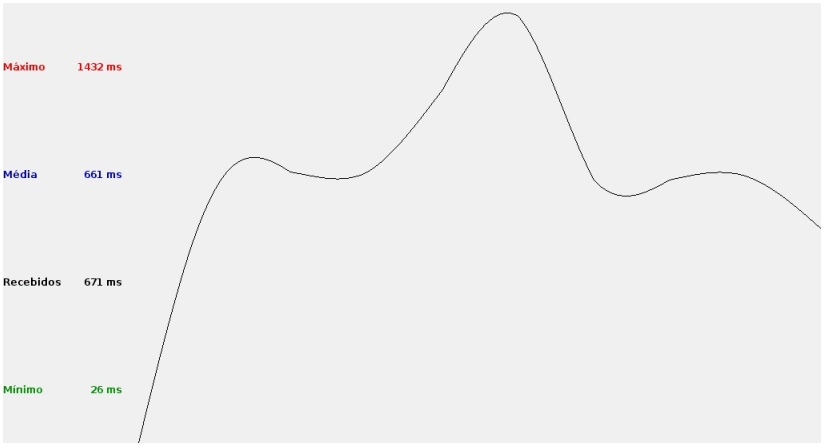

Figura 4.5: Visualizador Spline (Inserção de Dados). Fonte: Autor

Na visualização *Spline*, ainda é possível ver o tempo de resposta da requisição mais rápida (26 ms) e o tempo de resposta da requisição mais lenta (1432 ms).

4.4.2 Consulta de Limite Fixo

No teste de consulta de limite fixo, foi feita uma consulta de uma coleção de documentos, porém com o parâmetro `limit` fixo em 10. Ou seja, cada requisição retornou dez documentos inseridos no teste anterior por esta consulta. A figura 4.6 mostra o gráfico de resultados traçado pela ferramenta neste teste.

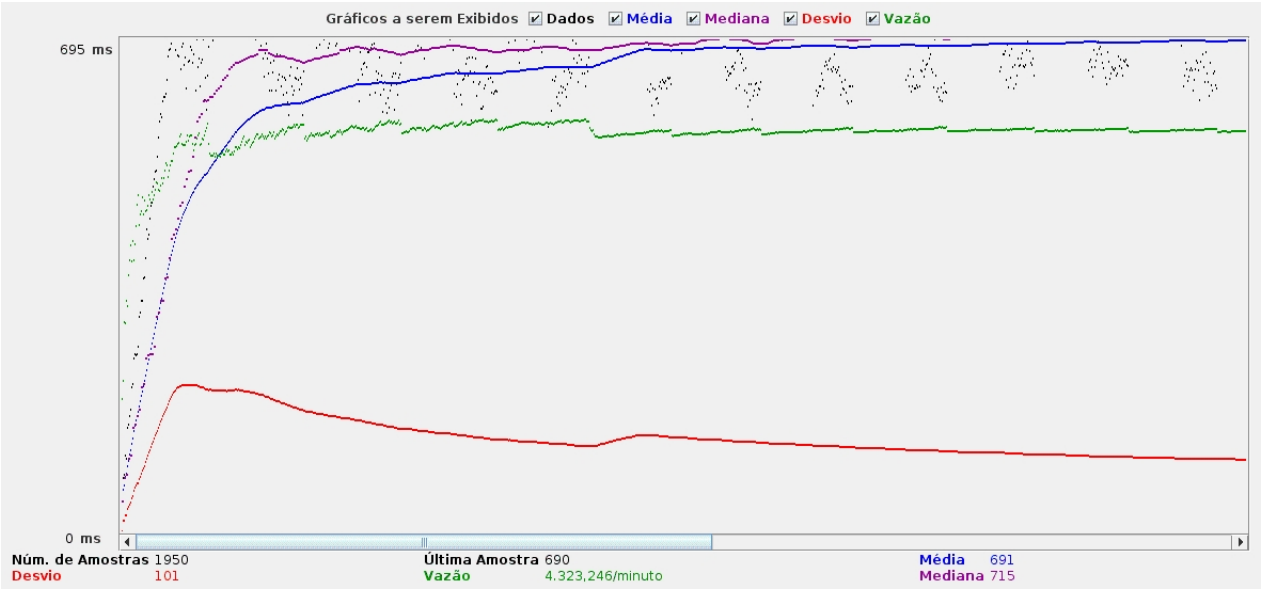

Figura 4.6: Gráfico de Resultados (Consulta de Limite Fixo). Fonte: Autor

Como mostra a figura, os dados coletados a partir do gráfico foram de número de amostras (1950), taxa de desvio (101 ms), tempo da última amostra (690 ms), taxa de vazão (4.323.246 req./min), média (691 ms) e mediana (715 ms). A figura 4.7 mostra a visualização *Spline* do mesmo teste de desempenho.

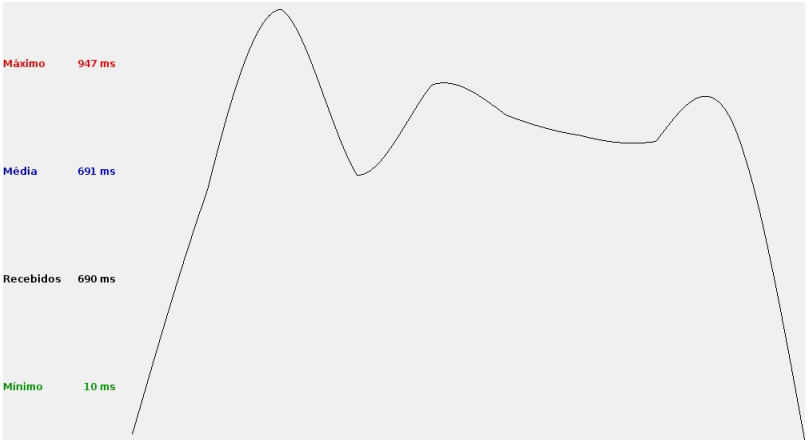

Figura 4.7: Visualizador Spline (Consulta de Limite Fixo). Fonte: Autor

Na visualização *Spline*, ainda é possível ver o tempo de resposta da requisição mais rápida (10 ms) e o tempo de resposta da requisição mais lenta (947 ms).

4.4.3 Consulta Unitária

No teste de consulta unitária, foi escolhido um identificador aleatório de um documento inserido no primeiro teste, e então foram feitas várias requisições para recuperar o mesmo documento. Ou seja, cada consulta retornou apenas um documento. A figura 4.8 mostra o gráfico de resultados traçado pela ferramenta neste teste.

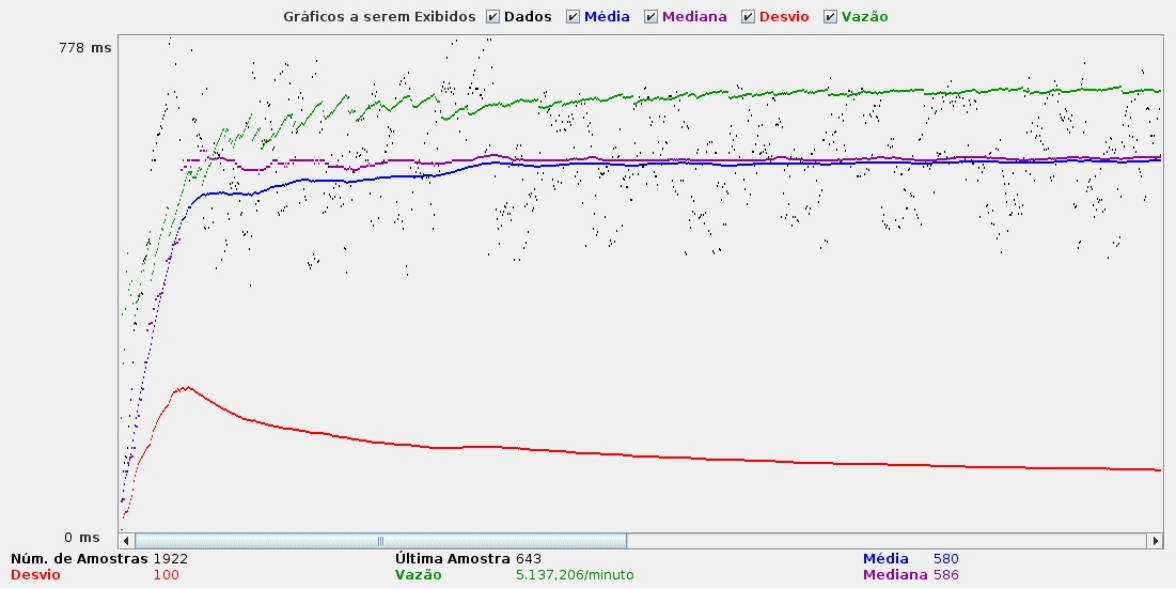

Figura 4.8: Gráfico de Resultados (Consulta Unitária). Fonte: Autor

Como mostra a figura, os dados coletados a partir do gráfico foram de número de amostras (1922), taxa de desvio (100 ms), tempo da última amostra (643 ms), taxa de vazão (5.137.206 req./min), média (580 ms) e mediana (586 ms). A figura 4.9 mostra a visualização *Spline* do mesmo teste de desempenho.

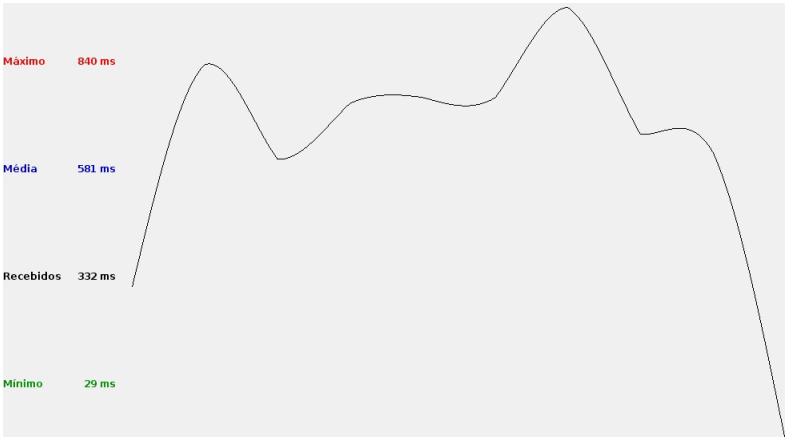

Figura 4.9: Visualizador Spline (Consulta Unitária). Fonte: Autor

Na visualização *Spline*, ainda é possível ver o tempo de resposta da requisição mais rápida (29 ms) e o tempo de resposta da requisição mais lenta (840 ms).

4.5 Avaliação Global da Solução

Neste capítulo foram abordados tópicos sobre a área de aplicação a qual o modelo proposto se encaixa, comparações de possíveis consultas pela interface e o SGBDR PostgreSQL e resultados dos testes de desempenho realizados em um ambiente de nuvem privada.

A aplicabilidade da solução é possível, visto que vários tipos de aplicações podem utilizar este modelo de solução, porém este sendo recomendado para aplicações web que se comportam melhor com o modelo de dados documental e que tem como requisito transações atômicas e consistência dos dados.

Um ponto forte da solução é a utilização de um modelo de dados NoSQL que pode ser consultado em conjunto com parte da linguagem SQL além de prover suporte à transações ACID, uma forte característica dos SGBDRs. Como ponto fraco pode-se citar a dificuldade de gerenciamento dos dados alocados em nós de maior nível na estrutura de árvore dos documentos, pois ainda não é possível criar índices para estes, limitando-se aos nós do primeiro nível.

Quanto aos resultados dos testes de desempenho, voltados para o tempo de resposta, mostram que ainda há uma barreira significativa, pois os testes de terceiros mostram as ferramentas com latência média abaixo de 100 milissegundos.

Porém a diferença de desempenho pode ser um defeito na arquitetura de nuvem implantada. O número de máquinas virtuais pode ser alterado de forma que o tempo de resposta ou mesmo a vazão da carga de trabalho sejam afetados. Para este tipo de problema existem outros métodos de solução, relacionados ao provisionamento de recursos para ambientes em nuvem (PFITSCHER; PILLON; OBELHEIRO, 2014).

Logo, a análise dos resultados obtidos a partir do teste de desempenho foi considerada satisfatória, apesar de carecer de aperfeiçoamento, para um ambiente de nuvem privada, onde a interface consolidou-se capaz de responder requisições para 300 de usuários com ausência de taxa de erro.

Estes resultados ainda podem sofrer possíveis melhorias com a implementação de um servidor web especializado para o uso da interface. Para este projeto foi utilizado o servidor web Apache em conjunto com o módulo *mod_wsgi*, mas existem várias outras formas de implementar um servidor web com *python*, inclusive melhores que o utilizado neste projeto, como aponta o *benchmark* feito por (PIEL, 2010).

Outra melhoria que pode ser feita é a remoção de bibliotecas utilizadas no projeto, como o framework *Pyramid* e o *SQLAlchemy*. Para uma primeira versão, esses *softwares* foram muito úteis para agilizar o processo de desenvolvimento, entretanto representam um atraso pois são ferramentas para aplicações genéricas, e possuem várias funcionalidades que não foram utilizadas.

Capítulo 5

Conclusão

5.1 Conclusões

Enquanto bancos de dados NoSQL são tecnologias em ascensão, e fundamentais para *startups* web, e os bancos de dados relacionais tecnologias maduras e bem difundidas no mercado, neste trabalho procurou-se beneficiar dessas duas tecnologias por meio de uma interface, partindo da ideia de que podem coexistir em uma tecnologia híbrida, de modo a possibilitar o gerenciamento de dados em nuvem.

Para atingir o objetivo específico da análise da arquitetura da computação em nuvem, foi apresentado um modelo em camadas da arquitetura, e constatou-se o escopo do projeto dentro do modelo de serviço SaaS e do modelo de implantação em nuvem privada.

Algumas técnicas de gerenciamento de dados em nuvem foram analisadas, e optou-se por utilizar a abordagem de compartilhamento total, onde inquilinos compartilham o esquema e a instância de banco de dados. A técnica de virtualização foi utilizada nos testes, onde foi configurada uma máquina virtual e replicada na rede para escalabilidade de processamento horizontal. A interface ficou tipificada com transações ACID, herdada do modelo relacional.

As arquiteturas e características dos modelos de bancos de dados não relacionais foram analisadas afim de identificar e avaliar vantagens e desvantagens de cada. A partir da análise, o modelo de dados documental se mostrou compatível com o modelo relacional e dentro das limitações de desenvolvimento adaptáveis em performance, escalabilidade, flexibilidade, complexidade e funcionalidade.

A arquitetura REST em conjunto com uma API permitiu integração com o modelo de dados, pois foi possível operacionalizar cada recurso por meio de métodos acessíveis por uma assinatura de rotas genérica criada para disponibilizar uma interface uniforme, permitindo também a comunicação via HTTP com clientes no ambiente de nuvem privada.

A linguagem SQL pôde ser parcialmente utilizada com a integração da interface ao SGBDR PostgreSQL, com apenas três modelos de tabelas e colunas personalizadas de acordo com a estrutura da base, estrutura criada neste trabalho para esquematização estrutural dos dados. Com esta implementação bastante simples, é possível integrar sem

muitas dificuldades a interface à outros bancos de dados relacionais.

A utilização do Python como linguagem de programação favoreceu a utilização de bibliotecas genéricas e especializadas que ajudaram no desenvolvimento rápido da parte de software do interface. Essa escolha aumenta consideravelmente a produtividade da tarefa de programação, pois o Python é uma linguagem para a qual há várias bibliotecas prontas (geralmente gratuitas e bem testadas) em diversas áreas diferentes, favorecendo a reutilização por meio de uma modularização de componentes de software.

Assim, o objetivo do trabalho foi atingido totalmente, pois foi possível reunir características dos modelos relacional e não relacional com a implementação do modelo de dados documental e o desenvolvimento de uma API integrada ao banco relacional PostgreSQL, e os testes da ferramenta foram feitos no ambiente de nuvem privada. Além disso, com a estrutura da base desenvolvida, foi possível conter a inconsistência dos dados, diminuindo a complexidade de consultas por causa da estrutura fixa dos documentos, e consequentemente, facilitando o desenvolvimento das aplicações.

5.2 Sugestões para Trabalhos Futuros

Durante o desenvolvimento deste projeto, algumas possibilidades de trabalhos futuros surgiram. A seguir citam-se cinco sugestões.

- Uma possibilidade de trabalho futuro é a evolução do modelo de tabelas utilizado neste trabalho. A limitação da interface com relação ao gerenciamento de dados em nós de maior nível pode ser reduzida com a criação de novas tabelas no momento da criação da base referentes às estruturas de grupos multi valorados apresentadas por este trabalho.
- Este trabalho não cobriu aspectos referentes à segurança da aplicação nem ao controle de usuários, sendo estes módulos que podem ser desenvolvidos e integrados à interface em um trabalho futuro. Para o desenvolvimento destes módulos, recomenda-se apurar com cautela assuntos relacionados ao desempenho do sistema.
- Além da escalabilidade de processamento, seria possível também a implementação da escalabilidade de armazenamento em trabalhos futuros. No projeto aqui desenvolvido, por não possuir nenhum tipo de chave estrangeira, este problema foi facilitado. Além disso, foram citadas algumas fontes sobre o assunto onde são tratadas questões sobre escalabilidade em SGBDRs.
- Uma interface gráfica é uma outra possibilidade de trabalho futuro. Pode ser desenvolvido um segundo projeto para administração visual do banco de dados, e posteriormente para que usuários possam modelar e gerenciar melhor seus dados.
- Por último, a extensão *plv8*, uma *engine javascript* criada pelo *Google*, poderia ser usada para implementar funções diretamente no PostgreSQL, aumentando drasticamente o desempenho da aplicação, pois o processamento seria feito pelo próprio banco de dados.

Referências Bibliográficas

ABADI, D. J. **Data Management in the Cloud: Limitations and Opportunities.** *IEEE Data Eng. Bull.*, v. 32, n. 1, p. 3–12, 2009.

AGRAWAL, D. et al. **Data management challenges in cloud computing infrastructures**. In: *Databases in Networked Information Systems*. [S.l.]: Springer, 2010. p. 1–10.

ALECRIM, E. ***Conhecendo o Servidor Apache (HTTP Server Project)***. [S.l.]: Unidavi, 2010.

AMAZON. ***Amazon Elastic Cloud Computing***. 2014. Aws.amazon.com/ec2/.

AN, A. **Application programming interface**. 2011.

APACHE. ***Apache***. 2014. Http://www.apache.org/.

ATZENI, P.; BUGIOTTI, F.; ROSSI, L. ***Uniform access to NoSQL systems***. [S.l.]: Elsevier, 2014. 117–133 p.

BREWER, E. **Pushing the CAP: Strategies for consistency and availability**. *Computer*, IEEE Computer Society Press, v. 45, n. 2, p. 23–29, 2012.

BUYYA, R. et al. **Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility**. *Future Generation computer systems*, Elsevier, v. 25, n. 6, p. 599–616, 2009.

CASSANDRA. ***Apache Cassandra NoSQL Performance Benchmarks***. 2014. Http://planetcassandra.org/nosql-performance-benchmarks/.

CHANG, F. et al. **Bigtable: A distributed storage system for structured data**. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 26, n. 2, p. 4, 2008.

CHEN, Q.; HSU, M.; WU, R. **MemcacheSQL a scale-out sql cache engine**. In: *Enabling Real-Time Business Intelligence*. [S.l.]: Springer, 2012. p. 23–37.

COMER, D. **Ubiquitous B-tree**. *ACM Computing Surveys (CSUR)*, ACM, v. 11, n. 2, p. 121–137, 1979.

COUCHBASE. ***Why Nosql?*** 2014. Http://www.couchbase.com/why-nosql/nosql-database.

COUCHDB. ***Apache CouchDB***. 2014. Http://couchdb.apache.org/.

COUTINHO, E. et al. **Elasticidade em computaçao na nuvem: Uma abordagem sistemática**. *XXXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2013)-Minicursos*, 2013.

DARROW, B. ***MongoDB or MySQL? Why not both?*** 2013. Https://gigaom.com/2012/05/25/mongodb-or-mysql-why-not-both/.

ELMORE, A. J. et al. **Towards an elastic and autonomic multitenant database**. In: *Proc. of NetDB Workshop*. [S.l.: s.n.], 2011.

FIELDING, R. T.; TAYLOR, R. N. **Principled design of the modern Web architecture**. *ACM Transactions on Internet Technology (TOIT)*, ACM, v. 2, n. 2, p. 115–150, 2002.

FOTACHE, M.; COGEAN, D. **NoSQL and SQL Databases for Mobile Applications. Case Study: MongoDB versus PostgreSQL**. *Informatica Economica*, Academy of Economic Studies-Bucharest, Romania, v. 17, n. 2, p. 41–58, 2013.

GAJENDRAN, S. K. ***A survey on nosql databases***. [S.l.], 2012.

GEERT, J. ***The Job of the API Designer***. 2010.

GOGRID. ***Scaling Your Internet Business***. 2010.

GOOGLE. ***Google App Engine***. 2014. Https://appengine.google.com/.

GOOGLE. ***GoogleDocs***. 2014. Https://docs.google.com/.

GREGOL, R. E. W. **Recursos de escalabilidade e alta disponibilidade para aplicações web**. Medianeira, 2012.

GRIGORIK, I. ***Tokyo Cabinet: Beyond Key-Value Store***. 2009.

GROLINGER, K. et al. **Data management in cloud environments: NoSQL and NewSQL data stores**. *Journal of Cloud Computing: Advances, Systems and Applications*, Springer, v. 2, n. 1, p. 22, 2013.

HEROKU. ***Heroku Cloud Application Platform***. 2014. Https://www.heroku.com.

IBM. ***IBM DB2 database software***. 2014. Http://www-01.ibm.com/software/data/db2.

JACKSON, J. **CouchBase, SQLite launch unified NoSQL query language**. *Accessed January*, v. 25, p. 2012, 2011.

JACOBS, D.; AULBACH, S. **Ruminations on Multi-Tenant Databases.** In: *BTW*. [S.l.: s.n.], 2007. v. 103, p. 514–521.

JMETER. ***Apache Jmeter***. 2014. Http://jmeter.apache.org/.

JONES, M. T. **Understand Representational State Transfer (REST) in Ruby**. *Developer Works*, IBM, p. 9, 2012.

JOSHI, A.; HARADHVALA, S.; LAMB, C. **Oracle nosql database-scalable, transactional key-value store**. In: *IMMM 2012, The Second International Conference on Advances in Information Mining and Management*. [S.l.: s.n.], 2012. p. 75–78.

JSON. ***Introducing JSON***. 2014. Http://json.org/.

LOWERY, J. C. ***Scaling-out with Oracle grid computing on Dell hardware***. 2003.

LUTZ, M. ***Learning python***. [S.l.]: "O'Reilly Media, Inc.", 2013.

MACHADO, F. S. e J. **Gerenciamento de Dados em Nuvem**. In: CSBC. *JAI-Jornadas de Atualização em Informática XXXIV Congresso da Sociedade Brasileira de Computação*. [S.l.], 2014.

MARCUS, A. **The NoSQL Ecosystem**. *The Architecture of Open Source Applications*, p. 185–205, 2011.

MELL, P.; GRANCE, T. **The NIST definition of cloud computing**. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011.

MICROSOFT. ***Microsoft Azure***. 2014. Azure.microsoft.com/.

MICROSOFT. ***Microsoft SQL Server***. 2014. Http://www.microsoft.com/en-us/server-cloud/products/sql-server.

MILLER, J. J. **Graph Database Applications and Concepts with Neo4j**. 2013.

MONGODB. ***Introduction to MongoDB***. 2014. Http://www.mongodb.org/about/introduction/.

MYSQL. ***The world's most popular open source database***. 2014. Http://www.mysql.com/.

NÄSHOLM, P. **Extracting Data from NoSQL Databases-A Step towards Interactive Visual Analysis of NoSQL Data**. Chalmers University of Technology, 2012.

NEO4J. ***Nosql Data Models***. 2014. Http://www.neo4j.org/learn/nosql.

NGINX. ***Nginx***. 2014. Http://nginx.com/.

OPENSTACK. ***OpenStack Open Source Cloud Computing Software***. 2014. Www.openstack.org/.

ORACLE. ***Oracle Database***. 2014. Http://www.oracle.com/Database.

PFITSCHER, R. J.; PILLON, M. A.; OBELHEIRO, R. R. **Diagnóstico do provisionamento de recursos para máquinas virtuais em nuvens IaaS**. *31 Simpósio Brasileiro de Redes de Computadores (SBRC)*, p. 599–612, 2014.

PIEL, N. ***Benchmark of Python WSGI Servers***. 2010. Http://nichol.as/benchmark-of-python-web-servers.

POSTGRESQL. ***The world's most advanced open source database***. 2014. Http://www.postgresql.org/.

PRITCHETT, D. **Base: An acid alternative**. *Queue*, ACM, v. 6, n. 3, p. 48–55, 2008.

PYRAMID. ***The Pylons Project***. 2014. Http://www.pylonsproject.org/.

RIAK. ***Riak Basho Technologies***. 2014. Http://basho.com/riak/.

SALESFORCE. ***Salesforce***. 2014. Http://www.salesforce.com/.

SHARMA, U. et al. **A cost-aware elasticity provisioning system for the cloud**. In: IEEE. *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*. [S.l.], 2011. p. 559–570.

SOROR, A. A. et al. **Automatic virtual machine configuration for database workloads**. *ACM Transactions on Database Systems (TODS)*, ACM, v. 35, n. 1, p. 7, 2010.

STRAUCH, C.; SITES, U.-L. S.; KRIHA, W. **NoSQL databases**. *Lecture Notes, Stuttgart Media University*, 2011.

TANENBAUM, A. S.; STEEN, M. V. ***Sistemas Distribuídos. Princípios e Paradigmas.*** [S.l.]: Editora Pearson Prentice Hall, 2007.

TAYLOR, M.; GUO, C. J. **Data Integration and Composite Business Services, Part 3: Build a multi-tenant data tier with access control and security**. *Internet Article,[Online], Dec*, v. 13, p. 1–16, 2007.

TIWARI, S. ***Professional NoSQL***. [S.l.]: John Wiley & Sons, 2011.

TOTH, R. M. ***Abordagem NoSQL–uma real alternativa***. 2014.

VECCHIOLA, C.; CHU, X.; BUYYA, R. **Aneka: a software platform for .NET-based cloud computing**. *High Speed and Large Scale Scientific Computing*, IOS Press, Amsterdam, Netherlands, p. 267–295, 2009.

WHITTAKER, G. L. S. T. J. **Improving performance of schemaless document storage in PostgreSQL using BSON**. 2013.

WILDER, B. ***Cloud Architecture Patterns: Using Microsoft Azure***. [S.l.]: "O'Reilly Media, Inc.", 2012.

ZERIN, I. 2013. Http://www.mscs.mu.edu/ izerin/DB2013/Final.html.