Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Investigando o Uso de Bancos de Dados não Convencionais para Gerenciar Informações da Administração Pública

Diogo Araújo Pacheco Wanzeller

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Rodrigo Bonifacio de Almeida

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifacio de Almeida (Orientador) — CIC/UnB
Prof. Dr. Professor I — CIC/UnB
Prof. Dr. Professor II — CIC/UnB

CIP — Catalogação Internacional na Publicação

Wanzeller, Diogo Araújo Pacheco.

Investigando o Uso de Bancos de Dados não Convencionais para Gerenciar Informações da Administração Pública / Diogo Araújo Pacheco Wanzeller. Brasília : UnB, 2013.
123 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. big-data, 2. NoSql, 3. Banco de Dados

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Investigando o Uso de Bancos de Dados não Convencionais para Gerenciar Informações da Administração Pública

Diogo Araújo Pacheco Wanzeller

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Rodrigo Bonifacio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Professor I
CIC/UnB

Prof. Dr. Professor II
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Curso de Computação — Licenciatura

Brasília, 28 de janeiro de 2013

Dedicatória

Dedico a....

Agradecimentos

Agradeço a....

Abstract

A ciência...

Palavras-chave: big-data, NoSql, Banco de Dados

Abstract

The science...

Keywords: big-data, NoSql, Data Bases

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A sociedade está lidando com uma quantidade de dados cada vez maior. Conforme Borkar et al. [16] descreve, as empresas passaram a monitorar compras de clientes, pesquisas de produtos, sites de relacionamento e diversas outras fontes para aumentar a eficácia do seu marketing e dos serviços ofertados aos clientes; governos e empresas estão rastreando conteúdos de *blogs* e *tweets* para realizar análises de sentimentos; e organizações públicas de saúde estão monitorando artigos de notícias, *tweets*, e tendências de pesquisas na web para acompanhar o progresso de epidemias [16]. Esse grande volume de dados disponíveis e gerenciados leva a muitos desafios tanto para a academia quanto para a sociedade em geral.

Importante destacar que a definição de "grande volume de dados" evoluiu significativamente. Há pouco tempo, por exemplo, o armazenamento na ordem de *terabytes* era algo restrito a poucos domínios de aplicação (como os domínios de telefonia e financeiro). Por outro lado, atualmente a maior parte dos dispositivos de armazenamento alcançam capacidades superiores a um *terabyte* e empresas dos mais variados segmentos já gerenciam volumes de dados da ordem de *petabytes*. Isso também inclui órgãos da administração pública, que atualmente mantem a documentação do funcionalismo público em pastas físicas e que precisam ser armazenados com qualidade e cuidado, pois fazem parte dos chamados arquivos permanentes. Esses arquivos ocupam cada vez mais espaço e, devido a sua característica, devem ser preservados por um longo período de tempo. Por outro lado, inúmeras vezes os órgãos precisam consultar esses arquivos, em processos que consomem tempo, são difíceis de se realizar e contribuem para a deteriorização dos documentos [9].

A dificuldade em manter fisicamente esses documentos fez com que o governo federal incentivasse a administração pública a iniciar um processo de digitalização dos documentos [17] para que uma cópia digital desses arquivos fosse mantida pelos órgãos[1]. A digitalização dos arquivos não só possibilita a preservação dos documentos, pois restringe o manuseio dos originais, quanto também facilita o acesso, já que torna mais efetivo os acessos locais, remotos e/ou simultâneos. O processo de digitalização é complexo, demorado e, além de um controle de *work flow* bem definido, necessita de grandes investimentos de *software* e *hardware* para que o resultado tenha uma boa qualidade [9]. Em linhas gerais, o conceito de documentos descentralizados em pastas funcionais físicas será substituído por repositórios de dados e informações de origem primária, auditáveis e

[1]Mais informações sobre essa iniciativa podem ser encontradas na Portaria Normativa MP 3, de 18 de Novembro de 2011

não replicados. Isso caracteriza o Projeto de Assentamento Funcional Digital – AFD, que objetiva a criação de um *dossiê*, em mídia digital, que será tratado como Fonte Primária de Informação de dados cadastrais do Servidor Público Civil Federal e que substituirá a tradicional Pasta Funcional ou Assentamento Funcional. No site do SIGEPE (Sistema de Gestão de Pessoas) [10] são destacados alguns pontos de melhoria com a criação do AFD, destacando que "*A criação do Assentamento Funcional Digital (AFD) possibilitará a diminuição drástica do volume de papeis armazenados e tramitados. O AFD constituirá de um banco referencial, de dados e imagens das pastas funcionais, com indexadores para localização dos documentos de maneira online*" [3].

Para gerenciar grandes volumes de dados (caracterizando ambientes de *big data*), Podemos dividir as tecnologias em duas classes: as tecnologias envolvidas com análise dos dados, como *Hadoop* e o modelo de computação *MapReduce*; e as tecnologias de armazenamento eficiente para grandes volumes de dados [34], cujos avanços recentes levaram ao surgimento dos bancos de dados *NoSQL* (*Not only SQL*), com inovações relacionadas não apenas ao armazenamento mas também a distribuição de dados. Em linhas gerais, os Sistemas Gerenciadores de Bancos de Dados (SGBDs) relacionais não garantiam o tempo de resposta e escalabilidade esperados para ambientes de *big data*, fazendo com que os modelos *não relacionais* implementados por bancos de dados *NoSQL* passassem a ter uma aceitação crescente.

1.1 Objetivos e Justificativa

O principal objetivo desse trabalho é comparar os modelos relacional e não relacional de armazenamento para o contexto do Assentamento Digital Funcional (AFD). Mais especificamente, como existem diferentes alternativas de armazenamento em bancos de dados não relacionais, esse trabalho utiliza o modelo orientado a documentos como representante da *classe* de bancos de dados não relacionais.

Para atingir o objetivo principal, tínhamos os seguintes objetivos específicos:

- compreender, abstrair e modelar os conceitos e operações do AFD.
- modelar os conceitos do AFD utilizando a estratégia relacional.
- modelar os conceitos do AFD utilizando a estratégia orientada a documentos.
- implementar os modelos em SGBDs relacionais e orientados a documentos.
- implementar uma arquitetura SOA (*Service Oriented Architecture*)para realizar as operações do AFD, utilizando para persitência tando um SGBD relacional quanto um SGBD não relacional.
- projetar, implementar e realizar testes de desempenho considerando operações e volumes de dados que permitam tirar conclusões sobre quais dos modelos são mais propícios para o armazenamento dos dados do AFD.

Para cumprir esses objetivos, a arquitetura proposta para sustentar os testes consiste em serviços [22] com capacidades simples de inserção, consulta, exclusão e atualização de dados; utilizando uma camada de persistência implementada em diferentes SGBDs. Os

testes de desempenho da solução usam um banco de dados relacional (PostgresSQL) ou um banco de dados orientado a documentos (MongoDB).

Essa investigação se justifica porque, para que a base de dados do AFD possa cumprir com o seu propósito, ela precisa garantir bom tempo de resposta e escalabilidade.

1.2 Organização do Documento

Capítulo 2

Big Data, NoSQL e Testes de Software

Este capítulo traz uma contextualização sobre três temas relevantes ao trabalho. *Big Data* e *NoSQL* são os alicerces tecnológicos que motivaram nossa investigação, que considera alternativas recentes de armazenamento, tipicamente encontradas em bancos de dados não relacionais, e que suportam grandes volumes de dados (características de ambientes de *Big Data*). Como o alvo da pesquisa está relacionado ao desempenho no processamento das requisições, contextualizamos a área de testes de software que tem como um dos alvos a análise de desempenho de sistemas.

2.1 Big Data

A quantidade de informação que está disponível para a humanidade é enorme e a medida que o conhecimento humano se expande, maior é a quantidade dessa informação que precisa ser armazenada e analizada. Além da quantidade, o fluxo e variedade dessas informações constantemente desafiam a indústria e a academia a medida em que a quantidade de *big data* aumenta exponencialmente. Essa seção apresenta uma definição detalhada de o que é *big data* e as tecnologias que apoiam esse domínio.

2.1.1 O que é Big Data?

Em um estudo divulgado em 2011 o tamanho do universo digital quebrou a barreira dos *zettabytes* e esse número está crescendo rapidamente [5]. Cientistas de diversas áreas estão vendo o grande potencial de conhecimento que se pode adiquirir pela análise a armazenamento de informação digital. Conforme já dito anteriormente o conceito de "grande (*big*)" evoluiu no decorrer da nossa história. Na década de 70, grande significava *kilobytes*; ao longo do tempo cresceu para *gigabytes* e em seguida, a *terabytes*. Atualmente já podemos dizer que grande varia *petabytes* até *exabytes* [16]. Contudo o conceito de *big data* não se dá somente por tamanho ou domínio, mas sim por um conjunto de características que o difere de uma base de dados comum.

Segundo o Gartner *big data* é definido, em geral, como uma massa de dados de grande volume, velocidade e variedade de informações que exigem formas inovadoras de processamento para maior visibilidade e tomada de decisão [23]. A maioria dos estudiosos compartilham dessa mesma definição e dizem que *big data* é caracterizado por no mínimo três V's. Volume, variedade e velocidade. [29, 33]

Tabela 2.1: Tabela de bytes

Nome	Tamanho	Abreviação
`Kilobyte`	10^3	KB
`Megabyte`	10^6	MB
`Gigabyte`	10^9	GB
`Terabyte`	10^{12}	TB
`Petabyte`	10^{15}	PB
`Exabyte`	10^{18}	EB
`Zettabyte`	10^{21}	ZB
`Yottabyte`	10^{24}	YB

Volume é a característica mais fácil de se perceber. Geramos enormes quantidades de dados todos os dias, e essa quantidade só tende a aumentar. Redes sociais, dispositivos móveis que guardam nossas informações, sites que armazenam nossas preferências, dispositivos de busca que indexam as páginas da web e a popularização da computação em nuvem nos colocam em uma época de grande volume de dados, uma época em que tudo é informação, tudo é valioso, tudo pode ser extraído. Cada dia fica mais comum grandes empresas terem de ligar com dados na ordem de *petabytes*. Variedade é outra característica que é de fácil percepção, pois os dados são de diversas naturezas como email, dados gerados por mídias sociais (*blogs*, Twitter, Youtube, Facebook, *Wikis*), documentos eletrônicos, apresentações, fotos, mensagens instantâneas, dados médicos, videos, etc. A característica de velocidade é explicada quando precisamos processar os dados praticamente em tempo real como em controle de tráfego, detecções de fraudes e propagandas dinâmicas na web. Os dados são cada vez mais usados para tomadas de decisão em tempo real [15].

Dada a problemática do armazenamento, ao se deparar com os limites de técnicas e ferramentas disponíveis, o mercado tratou de criar suas próprias soluções de gerenciamento de dados, em sua maioria não relacional. Usando a tecnologia apropriada, profissionais capacitados podem transformar grandes massas de dados em informações muito valiosas. Muitos sistemas comercias relacionais se dizem capazes de lidar com vários *petabytes* de base de dados (Greenplum, Netezza, Teradata, ou Vertica). Apesar dessa quantidade de dados atender a grande maioria das empresas, existem empresas de grande porte como o Google e o Facebook que não são atendidas e precisaram criar suas próprias soluções, além disso, sistemas *open source* como Postgres não tem o mesmo nível de escalabilidade que os comerciais [29].

2.1.2 Bases de dados relacionais

Quando pensamos em armazenamento de dados em SGBDs logo associamos essa idéia ao método tradicional que inclui bancos de dados como MySQL, PostgreSQL, modelagem relacional e esquemas de dados bem definidos. O modelo de dados relacional foi introduzido por Ted Codd, da IBM Research, em 1970, em um artigo que conseguiu atrair grande atenção devido a simplicidade e base matemática. Os SGBDs relacionais mais populares

atualmente são o DB2 e Informix Dynamic Server (IBM), o Oracle e Rdb (Oracle), o Sybase SGBD (Sybase) e o SQLServer e Access (Microsoft). Ainda temos os de código aberto como o MySQL e PostgreSQL.

O modelo relacional representa o banco de dados como uma coleção de relações 2.1. Uma relação é como se fosse uma tabela de valores ou um arquivo de registros. Cada tabela é formada por uma ou mais colunas de dados. Por sua vez, cada linha na tabela contém uma instância única de dado para as categorias de colunas definidas. No modelo relacional é possível criar conexões entre as tabelas e os campos e os formatos dos valores são bem definidos, ou seja, possui um schema de dados [20, 28].

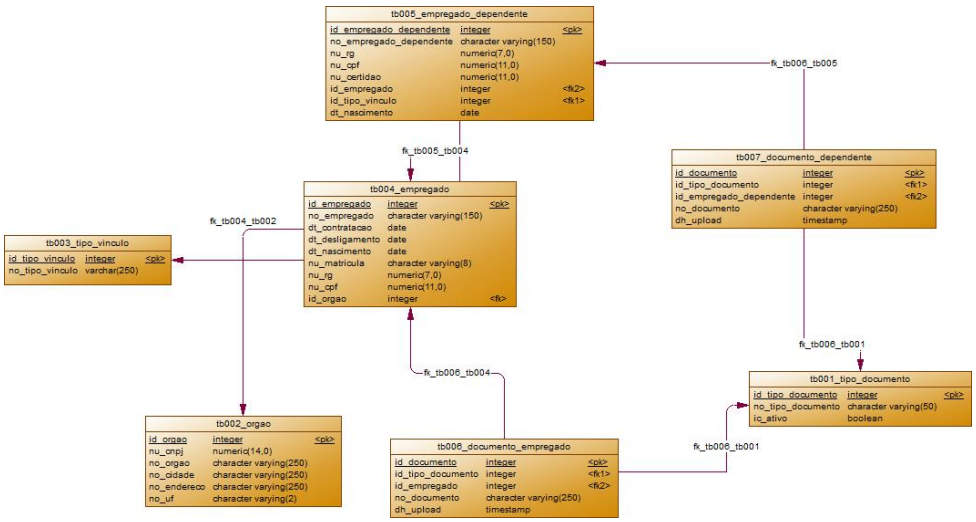

Figura 2.1: Modelo Relacional

Umas das características mais importantes das bases de dados relacionais são as garantias das propriedades ACID [31]. ACID é um acrônimo para *Atomicity*(Atomicidade), *Consistency*(Consitência), *Isolation* (Isolamento) e *Durability* (Durabilidade). Atomicidade significa que todas as etapas de uma transação serão executadas, caso contrário a transação será abortada sem interferir no banco de dados. Consistência significa que um banco de dados estará em um estado consistente antes e após cada transação. Caso as mudanças de uma transação violarem a regra de consistência, então todas as mudanças serão revogadas para garantir que somente os dados realmente válidos serão escritos no banco de dados. Isolamento, por sua vez, significa que as transações não podem visualizar as mudanças que não foram submetidas ao banco. Como as alterações que resultaram em erros. Finalmente, a propriedade de durabilidade requer que os dados sejam escritos no banco antes da transação ser confirmada. Caso haja uma falha de energia os dados não serão perdidos.

Os SGBDs (Sistemas Gerenciadores de Banco de Dados) relacionais proveem diversas garantias aos seus usuários, como validação, verificação e garantias de integridade dos dados, controle de concorrência, recuperação de falhas, entre outros. Todas essas características mantém os SGBDs como principal solução na maioria dos ambientes computacionais, mas não impediram o surgimento de problemas, em alguns casos, causados pela rígida estrutuda definida pelo *layout* das tabelas, nomes e tipos das colunas.

As abordagens mais usadas para manipular grandes bases nesse tipo de estrutura são os *data warehouses* e *data marts*. Um *data warehouse* é um banco de dados relacional usado para armazenar, analizar e gerar relatórios sobre os dados. O *data mart* é a camada usada para acessar o *data warehouse*. As duas abordagem usadas para se armazenar dados em um *data warehouse* são a normalização e a modelagem dimensional [14].

Por outro lado, com a evolução das aplicações e com requisitos cada vez mais exigentes, foram surgindo casos em que os banco de dados relacionais não escalavam. Operações de *joins* estão presentes nos menores dos bancos de dados relacionais, e esse tipo de operação é lenta. Para que SGBDs relacionais consigam garantir consistência para os dados eles usam o conceito de transações, o que requer um bloqueio nos dados durante um certo período de tempo. Dessa forma, quando o banco recebe várias requisições simultâneas em um mesmo dado os usuários são obrigados à esperarem em uma fila [26].

A necessidade de transformar os dados em tabelas causa um aumento na complexidade da operação pois requer o uso de complexos algoritmos de mapeamento e estrutura. Mesmo quando uma base de dados pode ser coberta pelo modelo relacional, as vezes as diversas garantias providas por esse modelo gera uma sobrecarga que não seria necessária para tarefas simples. O schema rigoroso pode ser pesado para aplicações que precisam de velocidade, como aplicações web e *blogs* que possuem diversos tipos de atributos. Textos, comentários, imagens, vídeos fonte, código e outras informações precisam ser armazenadas em diversas tabelas, e como as aplicações na web são muito ágeis, precisam ser amparadas por uma base de dados igualmente ágil e com um schema de fácil adaptação [25].

O considerável aumento na quantidade de dados deve ser considerado por grandes empresas como Facebook, Amazon e Google. Além de tratar *terabytes/petabytes* de dados, realizar requisições de leitura e escrita na base a todo o momento essas empresas devem se preocupar com o tempo que essas transações estão levando, ou seja, a latência. Para tratar esses requisitos é preciso manter milhares de máquinas com um *hardware* moderno e veloz. Por ter que cumprir com os requisitos de ACID e manter os dados normalizados, um modelo relacional não é adequado para esse cenário, visto que as operações de *join* bloqueiam os dados e influenciam negativamente no desempenho da aplicação.

Outro requisito fundamental para as grandes empresas é a disponibilidade de seus serviços. Para isso a base de dados deve ser facilmente replicável e fornecer uma forma automática de tratamento à falha de bases ou do *datacenter*. Esses SGBDs também devem ser capazes de balancear a carga em várias máquinas para não sobrecarregar um único servidor. Bancos relacionais priorizam a consistência em detrimento à disponibilidade e também possuem um mecanismo de replicação limitado.

Esses problemas podem ser resolvidos de algumas formas.Primeiramente optamos por um *upgrade* simples de *hardware*. Se o problema persistir a próxima opção seria adicionarmos novos servidores ao *cluster*, porém com os problemas de consistência e replicação durante o uso regular e em cenários de falha. A próxima etapa seria melhorar a configuração do gerenciador de banco de dados. Caso as opções de melhoria no SGBD se esgotem é preciso melhorar a aplicação. Verifica-se o desempenho das consultas, criamos índices e etc. Se o desempenho ainda não for satisfatório então talvez coloquemos uma camada de *cache*, mas que também gera um problema de consistência. Se mesmo assim o desempenho não atender as expectativas, então é necessário pensarmos novamente no SGBD. A última opção seria uma desnormarlização do banco, mas assim se estaria indo contra os princípios da modelagem relacional e das regras normais [26].

Dado toda essa problemática surge uma opção. Bancos de dados que não seguem o paradigma relacional. Os dados não são normalizados.

2.1.3 Bases Não Relacionais

O NoSQL foi proposto em 2009 e quebrou os limites das bases de dados relacionais e das propriedades ACID. Os banco de dados NoSQL geralmente não proveem as propriedades ACID: atualizações são eventualmente propagadas, mas há garantias limitadas para a consistência de leituras. Alguns autores sugeriram o acrônimo 'BASE' em contraste ao 'ACID' [19]. O teorema CAP, o teorema BASE e o conceito de consistência eventual são os fundamentos do NoSQL, discutidos no restante dessa seção.

Teorema CAP

O teorema CAP (Figura 2.2) foi proposto por Eric Brewer. CAP significa Consistência, disponibilidade e tolerância a particionamento de rede. A idéia principal desse teorema é que os sistemas distribuídos não podem atender, ao mesmo tempo, essas três características. Podem atender somente duas delas [36].

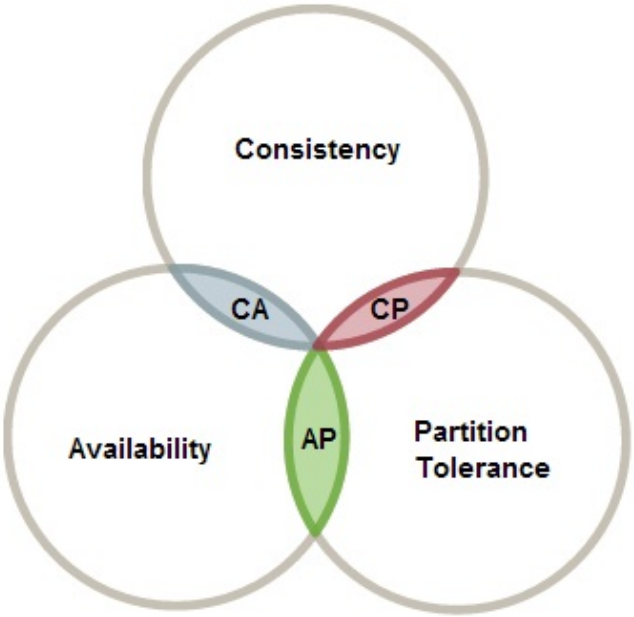

Figura 2.2: Teorema CAP [21]

Nesse caso a disponibilidade permite que os clientes sempre poderão executar leituras e escritas em um período de tempo. Um banco de dados distribuído que permite particionamento é tolerante a falhas de conexão e permite a distribuição em nós separados [31].

Um sistema que permite o particionamento só poderá prover uma consistência forte se sacrificar a disponibilidade. Isso porque cada operação de escrita somente será concluída

se os dados forem replicados para todos os nós, o que nem sempre é possível em um ambiente com falhas de conexão ou outras falhas de *hardware* [31].

Sendo assim, temos três arquiteturas possíveis: CA, AP e CP. Como atualmente a grande maioria dos sistemas é implantada na web, a disponibilidade é indispensável. Isso nos permite utilizar somente com as arquiteturas CA e AP. Para sistemas web, a disponibilidade e a tolerância ao particionamento são mais importantes que a consistência. Já é suficiente quando um sistema web possui consistência eventual [36].

Teorema BASE

O teorema BASE é um produto do teorema CAP. As propriedades BASE são completamente diferentes das ACID. BASE é um acrônimo para: *Basically Available* (Basicamente Disponível), *Soft-state*(base otimizada pelo uso) e *Eventual consistency* (Disponibilidade Eventual) [36].

- *BasicallyAvailable* : Significa que eventuais falhas de particionamento são suportadas.
- *Soft-state*: Significa que em um período de tempo o estado do sistema pode ser assíncrono.
- *Eventual consistency*: O sistema 'deve' ser consistente.

Consistência Eventual

Por causa o teorema CAP, a maioria dos banco de dados NoSQL proveem consistência eventual. Abaixo estão as definições de consistência forte e consistência eventual.

Consistência forte: Significa que todos os processos conectados em um banco de dados sempre verão a mesma versão de um valor e uma nova atualização é instantaneamente refletida por qualquer operação de leitura até outra mudança ser feita por outra operação de escrita [31]. Na Figura 2.3 temos um exemplo gráfico.

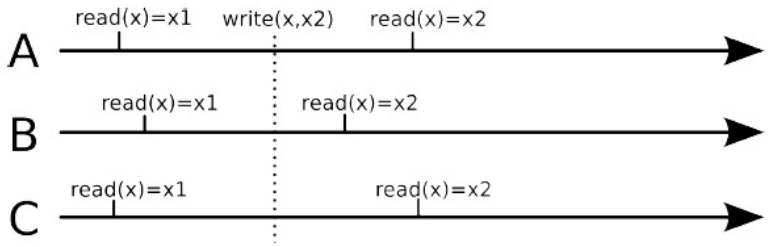

Figura 2.3: Consistência Forte: O processo A, B e C sempre estao vendo a mesma versão do banco de dados [31].

Consistência Eventual: É um tipo de consistência fraca. O não garante que todos os processos veem a mesma versão dos dados. Isso pode ocorrer por causa das janelas de inconsistência e é geralmente causada pela replicação dos dados nos diferentes nós [31]. Na Figura 2.4 temos um exemplo gráfico.

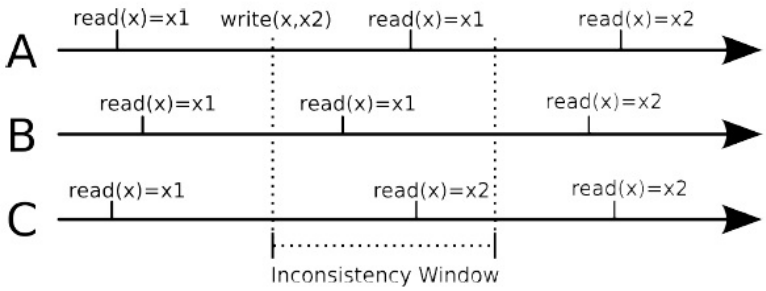

Figura 2.4: Consistência Eventual: Os processos A, B e C podem visualizar diferentes versões dos dados durante a janela de inconsistência causada pela replicação assíncrona [31].

As próximas seções discutem os modelos de dados NoSQL. Inicialmente é feita a definição do termo NoSQL e, em seguida, apresentamos diferentes formas de modelagem e armazenamento de bancos de dados que guiam as atuais alternativas ao modelo relacional. Apesar de discutirmos diferentes estratégias, o foco desse trabalho é comparar o modelo relacional, amplamente usado pela indústria desde o início da década de 90, com o modelo não relacional orientado a documentos. Por essa razão, a Seção 2.2.2 detalha o banco de dados orientado a documentos MongoDB.

2.2 NoSQL

O termo NoSql é a junção de duas palavras. *No* e *SQL*. Ao pé da letra significa que é uma tecnologia/produto que trabalha de forma contrária à tecnologia dos banco relacionais (adeptos do SQL). O termo é usado com o sentido de Não Relacional. Atualmente o termo NoSql é traduzido para *"Not only Sql"*, ou seja "Não só Sql". Saindo da definição, NoSql é um termo genérico para uma classe definida de banco de dados não-relacionais que armazenam os dados de forma diferente da conhecida modelagem relacional e que surgiram com o propósito de sanar algumas dificuldades encontradas com o modelo relacional. NoSql não é um produto, mas a uma classe de produtos e conceitos de armazenagem e manipulação de dados.

O que diferencia os bancos de dados NoSql dos relacionais são os seus modelos de dados sem um schema definido. Os bancos de dados NoSql podem ser classificados, segundo seu modelo de dados, em quatro grupos: chave-valor, orientados a documentos, orientados a colunas e baseados em grafos [18, 25].

Bancos de dados que usam a modelagem não relacionais não são novidades. Conforme discutido no livro *NoSql Professional* [35] eles surgiram junto com as primeiras máquinas de computar. Bases não relacionais ficaram conhecidas e cresceram por causa do uso de *mainframes* e em dominios específicos como o armazenamento de credenciais para autenticação. Esse NoSql que conhecemos hoje é uma nova visão, ou como diz Tiwari em [35], uma reencarnação que nasceu no mundo de aplicações web que necessitam de recursos escaláveis para tratar de sua enorme massa de dados. Apesar de o paradigma NoSql já ter

sido criado há algum tempo nenhum ele só tomou as proporções atuais depois que grandes empresas como Google, Amazon e Facebook começam a usar em suas arquiteturas [25].

Ao utilizar SGBD's relacionais com grandes quantidade de dados surgem problemas como falta de eficiência no processamento, uma paralelização não efetiva, alto custo e scalabilidade limitada. Sendo um gigante da Internet, a Google, se não for a empresa que manipula a maior quantidade de dados, é com certeza uma das maiores e ao se deparar com essa problemática construiu a sua própria infraestrutura para que o seu mecanismo de busca e outras aplicações pudessem tratar a massa de dados de forma eficiente.

Com o lançamento de artigos pelo Google que explicavam em partes como o problema foi solucionado, desenvolvedores de *software* livre criaram o primeiro motor de busca de código aberto que replicava algumas característica da infraestrutura do Google, o Lucene. Logo depois, os principais desenvolvedores do Lucene se juntaram ao Yahoo e com a ajuda de diversos outros desenvolvedores criaram uma estrutura que imitada todas as peças da infraestrutura de computação distribuida do Google. Essa solução livre é o Hadoop. Nessa mesma época surgiu a idéia do NoSql.

O sucesso do Google e o Hadoop ajudaram a impulsionar novos conceitos de computação distribuída, NoSql e o próprio projeto Hadoop. Um ano após o lançamento dos artigos do Google outra gigante da internet resolveu compartilhar o seu caso de sucesso. Em 2007 a Amazon mostrou ao mundo sua solução de base de dados distribuída, disponível e consistente que se chama Dynamo.

Após Google e Amazon mostrarem a aplicabilidade do NoSql, diversos outros produtos surgiram nessa linha. O NoSql e os conceitos de manipulação de *big data* ganharam espaço e forma surgindo diversos casos de uso de sucesso de grandes companias como o Facebook, Netflix, Yahoo, EBay, Hulu, IBM e diversas outras.

2.2.1 Modelos de Banco de Dados NoSql

Chave-Valor

Bancos de dados NoSql que usam a modelagem Chave-Valor armazenam os dados indexados por um valor chave. A base é similar a um dicionário, onde os dados são endereçados por uma única chave. Uma vez que os dados são armazenados, é através das suas chaves a única forma de recuperá-los. Os valores sao isolados e independentes um dos outros, sendo necessário tratar isso na aplicação. Por isso banco chave-valor são livres de schema. Isso permite que novos tipos de dados sejam inseridos em tempo de execução sem que o banco entre em conflito e sem influenciar na disponibilidade do sistema [25, 28].

Alguns exemplos de banco de dados que usam esse tipo de modelagem são: RIAK, LevelDB, Voldemort, redis [8].

Orientados a Documentos

A modelagem orientada a documentos armazena os dados encapsulados em pares de chave-valor em JSON ou em outro padão semelhante. Dentro dos documentos as chaves devem ser únicas. Cada documento recebe um identificador que também é único dentro de uma coleção de documentos. Os documentos são as unidades básicas e não têm uma estrutuda definida como nas tabelas do modelo relacional, ou seja, não tem um schema de dados definido. Ao armazenar os dados em JSON há uma vantagem adicional que é

o suporte a tipos de dados, o que torna a forma de armazenamento mais amigável para os desenvolvedores [18, 25]. O exemplo de codificação 2.1 nos mostra como é a estrutura desse tipo de banco de dados.

Os exemplos mais significativos são: CouchDB, MongoDB e Riak [25].

Listing 2.1: Exemplo de arquivo do CouchDB

```
{
    "Subject": "I like Plankton",
    "Author": "Rusty",
    "PostedDate": "5/23/2006",
    "Tags": ["plankton", "baseball", "decisions"],
    "Body": "I decided today that I don't like baseball. I like
       plankton."
}
```

Orientados a Colunas

Nesse tipo de modelagem o paradigma passa a ser de orientação a atributos(colunas). Ao contrário da modelagem chave-valor, agora os dados são armazenados usando tabelas sem um schema definido, mas sem suporte a associação entre elas . Figura 2.5 Segundo Jing Han et all, um banco orientado a colunas tem as seguintes caracteristicas [24]:

1. Os dados são armazenados em colunas
2. Cada coluna de dado é um índice do banco
3. Acessar somente colunas faz com que haja redução de I/O nos resultados das consultas
4. Consultas simultâneas, isto é, cada coluna é tratada por um processo
5. Possuem o mesmo tipo de dados, características semelhantes e boa taxa de compressão

Em geral esse tipo de banco é mais vantajoso para aplicações de agregação e data warehouses. Alguns exemplos são: Cassandra e Hypertable [8].

Baseados em Grafos

Nessa categoria os dados são armazenados em nós de um grafo cujas arestas representam o tipo de associação entre esses nós. Esse tipo de banco é especializado em manter dados fortemente ligados. O twitter armazena as relações entre os seus usuários no seu próprio banco de dados baseados em grafos, o FlockDB, que é otimizado para listas de relações muito grandes, leituras e escritas [25]. Alguns exemplos são: Neo4J, infoGrid e FlockDB [8].

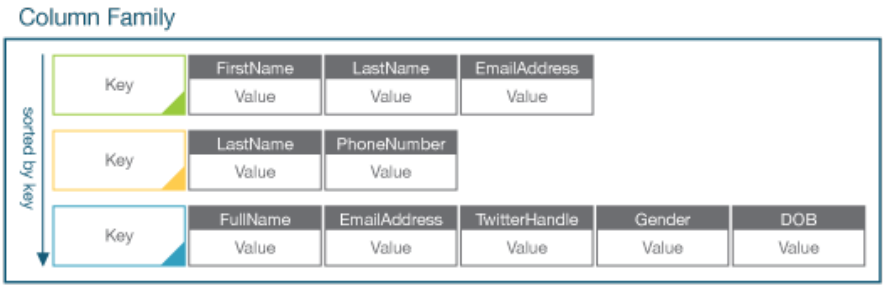

Figura 2.5: Modelagem orientada a colunas

2.2.2 MongoDB

Essa seção foi baseada no site oficial do MongoDB [7] exceto quando explicitamente citado. MongoDB é um banco de dados NoSQL, de código aberto, orientado a documentos, *schema-free* e escrito em C++. Os dados são persistidos em coleções de dados que são representados usando o BSON, um formato binário similar ao JSON (Figura 2.6). O MongoDB tem suporte a todos os tipos de dados JSON como string, inteiro, boleano, double, array e objeto. Por usar codificação BSON o MongoDB suporta alguns tipo de dados adicionais como data, binary data, regular expression e code [35]. Na Figura 2.1 podemos ver os tipos suportados pelo BSON.

Como não usa o mesmo formato de armazenamento dos SGBDS relacionais, o MongoDB armazena os seus dados em coleções, que são equivalentes às tabelas. Uma Coleção pode ter um ou mais documentos; são equivalentes as linhas em uma tabela de um banco de dados relacional. Cada documento tem um ou mais campos, o que corresponde a uma coluna.

Diferente do que a maioria das pessoas estão acostumadas, o MongoDB não trabalha com uma estrutura de dados bem definida (schema), ou melhor dizendo, ele usa schemas dinâmicos. Com ele é possível criar coleções sem que a estrutura, campos ou tipos de valores dos documentos estejam definidos. Essa forma flexível de armazenar os dados nos permite trabalhar com estruturas e dados bastante heterogêneos.

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

Figura 2.6: Documento BSON usado no MongoDB [7]

Tabela 2.2: BSON - Tipos Suportados

Tipo	Número
Double	1
String	2
Object	3
Array	4
Binary Data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

Quanto mais controle, mais custosa é uma operação para o sistema gerenciador de banco de dados. Os banco de dados NoSQL, como dito anteriormente, foram criados para suprir algumas características que os banco de dados relacionais não atendiam. Uma dessas características é a velocidade com que operações de consulta, escrita, atualização e exclusão são executadas. Para que a velocidade dessas transações fosse aumentada, foi preciso retirar alguns controles e, com isso, os banco de dados NoSQL não se comprometem com todas as características ACID.

O MongoDB não provê transações ACID, mas possui alguns recursos transacionais básicos. Operações atômicas são possíveis no escopo de um único documento. Na tabela abaixo temos alguns exemplos de operações em SQL e suas correspondentes no MongoDB.

Modelagem dos Dados

Cada documento tem um campo chamado ID que é utilizado como chave primária. Para aumentar a velocidade das *queries* é possível habilitar índices para os campos que são utilizados nas consultas. O MongoDB também suporta índices em sub-documentos e em arrays.

Ao contrário dos bancos de dados convencionais o MongoDB possui um schema flexível e não nos força a determinar uma estrutura antes de inserir os dados. As coleções no MongoDB não nos impedem de evoluir a estrutura dos documentos [31].

Para representar as relações entre os objetos temos duas estratégias: referências e sub-documentos [31].

O referenciamento representa as relações entre os dados incluindo *links* ou referências de um documento para outro. Para acessar os dados referenciados a aplicação deve resolver a referência. Na Figura 2.7, temos um exemplo de modelagem utilizando referência.

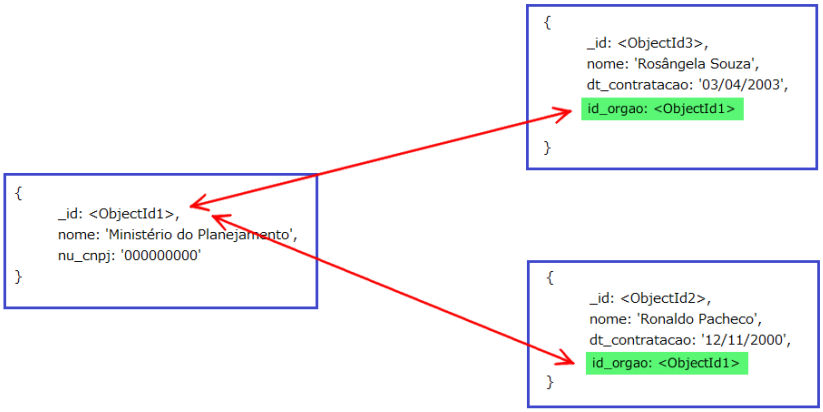

Figura 2.7: Modelagem utilizando referência. Adaptado de [7]

Seguem algumas ocasiões em que podemos utilizar referências como estratégia de modelagem:

- Quando, ao criar sub-documentos, criamos duplicação de dados e essa duplicação não nos dá um ganho de performance vantajoso.
- Quando desejamos representar complexas relações de n para n.
- Para representar um grande volume de dados de forma hierárquica.

Ao implementar as referências temos mais flexibilidade se compararmos com os sub-documentos. Em contrapartida, ao realizarmos consultas, essas referências deverão ser traduzidas, o que gera mais consultas ao servidor [31].

A outra estratégia possível é o uso de sub-documentos. Ao utilizar essa forma de armazenamento os dados relacionados são armazenados em um único documento. O MongoDB permite armazenar essas relações em sub-documentos ou arrays de documentos. Veja um exemplo desse tipo de modelagem na Figura 2.8.

Seguem algumas ocasiões em que podemos utilizar os sub-documentos:

- Em relações um para um;
- Em relações um para muitos. Nesses relacionamentos o objeto que se repete deve estar contido no outro objeto;

Os sub-documentos aumentam a performance de operações de leitura e nos dá a vantagem de obter os dados necessários em uma simples consulta ao banco. Com os sub-documentos também é possível realizar atualizações de forma atômica [31].

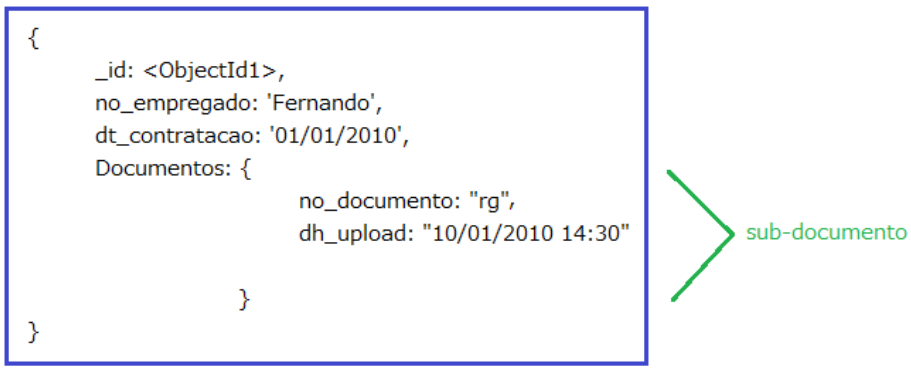

Figura 2.8: Modelagem utilizando sub-documento. Adaptado de [7]

Linguagem de Consulta

A sintaxe da linguagem de consulta do MongoDB é similar ao JSON. A linguagem de consulta permite consultar todos os documentos em uma coleção, inclusive os sub-documentos e os arrays [31].

A linguagem de consulta suporta [31]]:

- Consultas em documentos e sub-documentos
- Comparações
- Operações lógicas
- Ordenação por múltiplos campos
- *Group by*
- Uma agregação por consulta

Adicionalmente o MongoDB permite realizar consultas com agregações mais complexas utilizando uma variação do MapReduce [31].

Nas tabelas 2.3 e 2.4 temos algumas comparações entre a linguagem utilizada pelo MongoDB e o SQL.

Replicação

Replicação é o processo de sincronizar dados através de diferentes servidores. Com a replicação é possível prover redundância e aumentar a disponibilidade dos dados, além de proteger os dados de uma possível falha de *hardware* ou catástrofes.

Um conjunto de réplicas é um grupo de instâncias do MongoDB com os mesmos dados. Na arquitetura de um conjunto de replicação somente um servidor, o primário, recebe todas as requisições de escrita. Os outros servidores somente replicam as operações em suas instâncias. A Figura 2.9 representa a arquitetura para replicação.

Como somente um nó recebe todas as operações de escrita, para suportar a replicação, o nó primário armazena em log todas as operações. Os nós secundarios replicam os logs do nó primário e, em seguida, realizam as operações em suas instâncias. Caso o nó primário fique indisponível, o conjunto de replicação eleje um novo nó para ser o primário. Por padrão, as requisições de leitura são feitas ao nó primário, porém isso pode ser alterado. Como a replicação é assíncrona, se as preferências de leitura forem alteradas os dados retornados podem não ser os mais atuais.

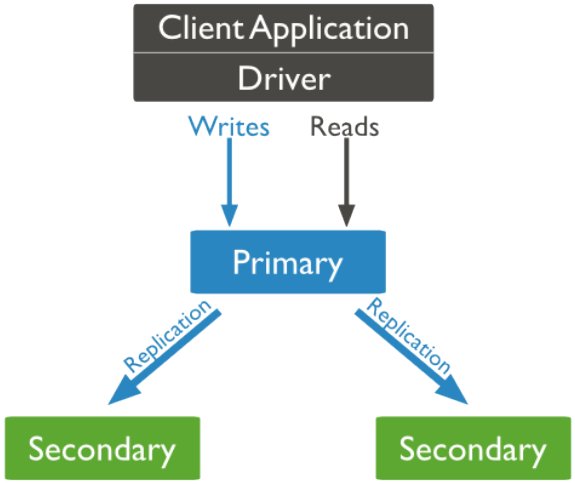

Figura 2.9: Arquitetura de Replicação [7]

Decomposição (Sharding)

Sharding, ou decomposição, é quando separamos a localização fisica dos dados, despedaçando cada informação e colocando-as em nós diferentes. Essa técnica é utilizada para trabalhar com dados de grande volume e com alta vazão de operações. A decomposição é a alternativa à scalabilidade vertical. No MongoDB a decomposição é implementada com o uso de um cluster de decomposição. Os componentes desse cluster são: *Shards*, rotiadores de consulta e servidores de configuração (Figura 2.10).

Os *shards* armazenam os dados.

Os rotiadores de consulta, ou mongos, se comunicam com os clientes e direcionam as operações ao *shard* ou *shards* apropriados.

Os servidores de conifguração armazenam os metadados do cluster. Ele contém o mapa do cluster. O roteador de consulta utiliza esse componente para encontrar os *shards* que serão utilizados.

Tratamento de Falhas

O MongoDB não utiliza log de transações para garantir a durabilidade dos dados. E por utilizar arquivos mapeados em memória, implementa escrita preguiçosa (*lazy write*). Sendo asssim, se um nó MongoDB falhar, provavelmente algum dado será perdido [31].

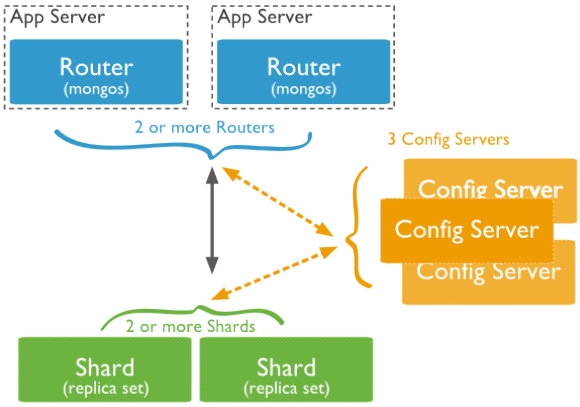

Figura 2.10: Cluster de Sharding [7]

2.3 Testes

Nessa seção veremos como o teste está diretamente ligado à qualidade de *software*, os principais tipos de testes, conceitos e ferramentas de automação, veremos quais os tipos de testes que serão aplicados ao nosso projeto e a importância da infra-estrutura de testes em um projeto.

2.3.1 Definição

Segundo o dicionário de termos da IEEE, teste é definido da seguinte forma:

- Teste: atividades nas quais um sistema ou um componente é executado sob determinadas condições e os resultados são observados ou gravados, e uma avaliação é feita observando determinado comportamento do sistema ou do componente;

2.3.2 Teste de Software e Qualidade de Software

O teste de *software* está diretamente ligado com a qualidade do *software* que está sendo desenvolvido. Podemos ver essa ligação já na definição de qualidade de *software*.

- Qualidade de *Software*: Conformidade a requisitos funcionais e de desenvolvimento explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido.

Dentro da qualidade de *software* temos a atividade de garantia de qualidade de *software*, e esta compreende uma variedade de tarefas associadas a sete grandes atividades, entre elas a atividade de testes:

1. Aplicação de métodos técnicos;
2. Realização de revisões técnicas formais;
3. Atividades de testes de *software*;

4. Aplicação de padrões;
5. Controle de mudanças;
6. Medição;
7. Manutenção de registros e reportagem;

Então podemos estar certos de que se queremos um *software* que atenda aos requisitos especificados, funcionais e não funcionais, que possua uma quantidade de erros reduzida e um desempenho que atenda ao usuário, uma tarefa que não pode ser despensada é o teste da aplicação. Como já dito anteriormente, teste de *software* e qualidade de *software* estão intimamente ligados, na tabela 2.5 podemos ver quais as características de qualidade são verificadas por determinados tipos de testes.

2.3.3 Tipos de Testes

O teste de *software* nos permite trabalhar com diversas estratégias e em diferentes níveis da aplicação. Emerson Rios e Trayahú Moreira [32] dizem que muitas vezes os tipos de *software* se sobrepõem, sendo até mesmo as suas definições abrangentes ou específicas, confome sua execução. Nessa seção listaremos os principais tipos de testes descritos por esses autores.

Testes Caixa Preta

Esse tipo de teste tem como objetivo verificar as funcionalidades da aplicação e a aderência aos requisitos, do ponto de vista do usuário, sem se basear no código ou lógica interna da aplicação.

Testes Caixa Branca

Os testes de caixa branca avaliam o código, a lógica interna do componente, as configurações e outros elementos técnicos.

Testes unitários

Esse é o tipo de teste que analisa o estágio mais baixo da aplicação. São aplicados nos menores componentes de código criados, verificando o atendimento as especificações e funcionalidades. Verificam o funcionamento de um pedaço do sistema, componente ou programa, isoladamente. Geralmente são realizados pelos próprios desenvolvedores.

Testes de integração

Esse teste visa testar se as interações estre os componentes da aplicação está resultando em algum tipo de erro. Tem como objetivo assegurar que as interfaces funcionem corretamente e que os dados são processados corretamente.Componentes podem ser pedaços de código, módulos, aplicações distintas, clientes e servidores

etc. Esse tipo de teste possui várias estratégias. Podemos testar a integração desde os componentes de mais baixo nível (*Booton-up*) até o sistema como um todo (Teste de sistema). Para o nosso trabalho nos atentaremos ao teste de sistema.

Testes de sistema

Esse teste é executado sobre o sistema como um todo, ou um subsistema, dentro de um ambiente operacional controlado. Deve ser simulada a operação normal do sistema, sendo testadas todas as suas funções de forma mais próxima possível do que irá ocorrer no ambiente de produção. É nesse estágio que deve-se realizar os testes de carga, performance, usabilidade, compatibilidade, segurança e recuperação.

Testes de aceitação

São realizados pelos usuários e visam garantir que a solução atenda aos objetivos do negócio e a seus requisitos, verificando as funcionalidades e a usabilidade do *software*.

Testes Back-to-back

É quando o mesmo teste é executado em versões diferentes do *software* e os resultados são comparados.

Testes de Configuração

É nesse tipo de teste de a execução da aplicação é analisada em diferentes configurações de ambiente.

Testes de Usabilidade

Mede a facilidade de uso da aplicação pelos usuários. É mais comum em aplicações web.

Testes de Segurança

Verifica o quão segura é a aplicação a acesso de usuários não autorizados.

Testes de Recuperação

Mede a qualidade da recuperação do *software* após falhas de *hardware* ou outro problemas inesperados.

Testes de Compatibilidade

Verifica se um *software* é capaz de ser executado em um ambiente determinado.

Testes de Desempenho

Verifica a adequação da aplicação aos níveis de desenpenho e tempo de resposta definidos nos requisitos. Também são conhecidos como testes de performance.

2.3.4 Testes de Carga e de Performance

Como o objetivo do trabalho é medir o desempenho da nossa aplicação com o uso de diferentes bancos de dados, restringimos os testes que serão usados no nosso projeto aos testes de carga e performance.

Testes de carga

Permite avaliar a aplicação sob uma alta carga de dados, repetidas entradas de dados, consultas complexas ou uma grande quantidade simultânea de usuários. Dessa forma é possível medir o nível de escalabilidade da aplicação. Esse tipo de teste deve ser aplicado durante os testes de sistema e também podem ser chamados de testes de estresse.

Teste de Performance

Molyneaux fala que do ponto de vista dos usuários, uma aplicação possui boa performance quando ela o permite realizar determinada tarefa sem demora [30]. Ela ainda diz que em uma aplicação performática o usuário nunca poderá se deparar com uma tela vazia ao realizar operações. O teste de performance é usado para medir o desempenho, em tempo de execução, e com todos os módulos integrados. Conforme Molyneaux, dividiremos os requisitos de performance em dois: orientados a serviço e orientados a eficiência.

Os indicadores de performance orientados a serviço são a disponibilidade e o tempo de resposta. Eles medem a qualidade do serviço que a aplicação está provendo ao usuário. Já os indicadores orientados a eficiência são a vazão e utilização. Vamos definir esses termos:

- Disponibilidade: É a característica de estar disponível para o usuário. Em softwares críticos, qualquer período de indisponibilidade pode gerar grandes prejuísos.
- Tempo de resposta: É o intervalo de tempo entre a requisição e a resposta da aplicação.
- Vazão: É a taxa em que os eventos da aplicação ocorrem.
- Utilização: É a porcentagem da capacidade total de recursos da aplicação que esta sendo usada.

Para que o nosso processo de teste de performance seja bem sucedido precisamos seguir algumas etapas.

1. Escolher uma ferramenta de teste de performance apropriada;
2. Desenvolver um ambiente de teste adequado a realidade dos testes e o mais próximo da realidade;
3. Escolher os objetivos que desejamos alcançar no trabalho;
4. Identificar e criar scripts para as transações críticas para o negócio;

2.3.5 Automação de Testes

Durante muito tempo os testes de *software* foram feitos manualmente. Os proprios programadores eram encarregados de simular as mais diversas situações [32]. Com o passar do tempo as aplicações se tornaram muito mais complexas e, consequentemente, o processo de teste manual se tornou inviável. Esse cenário foi ideal para que surgissem ferramentas de automação do processo de testes.

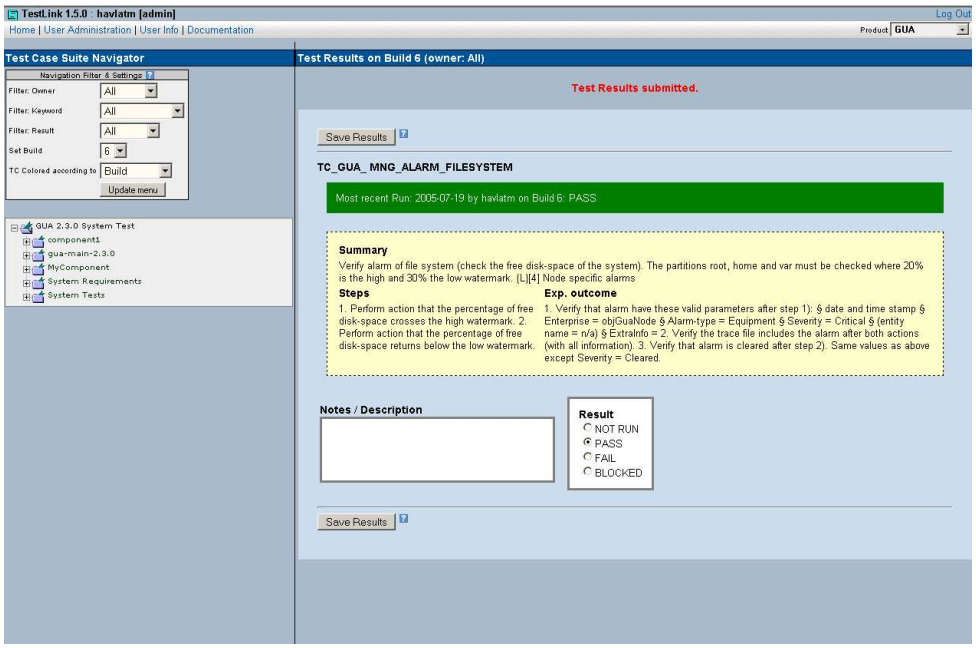

Figura 2.11: TestLink - acompanhamento/suporte [11]

As ferramentas de automação de teste visam facilitar o processo de teste e podem auxiliar no desenvolvimento dos testes, execução, manuseio das informações de resultado e a comunicação entre os envolvidos no processo. Utilizando *scripts* essas ferramentas são capazes de simular a utilização da aplicação por um ou vários usuários e, além disso, podem ser simulados vários cenários de uso. As ferramentas de teste podem ser divididas em três grupos: desenvolvimento, execução 2.12 e acompanhamento/suporte 2.11.

2.3.6 JMeter

O Apache JMeter é uma aplicação *open source*, 100% desenvolvida em Java e que foi criada para a execução de testes de carca e para medição de performance. Foi originalmente criado para testar aplicações web. O JMeter pode ser usado para testar a performance tanto de recursos estáticos quanto de recursos dinâmicos (arquivos, servelts, scripts Perl, objetos Java, Bancos de dados e queries, Servidores FTP e etc). Com ele é possível simular cargas pesadas em um servidor, rede ou objeto para testar o seu comportamento ou para analisar a performance em diferentes tipos de carga [2].

O JMeter pode testar diferentes tipos de servidores como:

- Web - HTTP, HTTPS

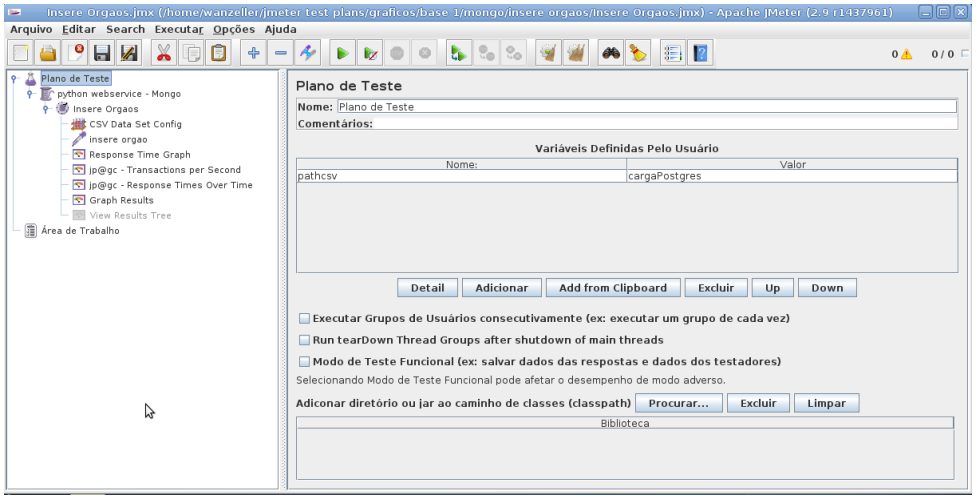

Figura 2.12: JMeter - Ferramenta para execução de testes [2]

- SOAP
- Database via JDBC
- LDAP
- JMS
- Mail - SMTP, POP3 e IMAP
- Comandos nativos ou scripts shell

Para realizarmos testes no JMeter precisamos criar um plano de teste. O plano de teste descreve uma série de passos que o JMeter terá de executar. O plano de teste pode conter os seguintes elementos: Grupo de Thread, controladores lógicos, testadores, ouvintes, *timers*, *assertions* e elementos de configuração. A seguir vamos ver os elementos que serão usados nos planos de teste desse trabalho.

Quando iniciamos o nosso plano de teste, o primeiro item que devemos procurar é o testador. Os testadores basicamente enviam requisições aos servidores e aguardam retorno. Cada testador possui diversas configurações que podem ser customizadas.

Testador de Requisição SOAP/XML - RPC

O testador SOAP (figura 2.13) é usado para mandar requisições SOAP para um Web service. Ele cria uma requsição HTTP POST com os dados especificados e executa o POST. As principais configurações são:

- **URL**: Endereço do WSDL do Web service.
- **Ação SOAP**: Endereço da requisição SOAP que o testador utilizará.
- **Dados SOAP/XML-RPC**: Requisição que será enviada para o Web service. Deve estar em formato XML.

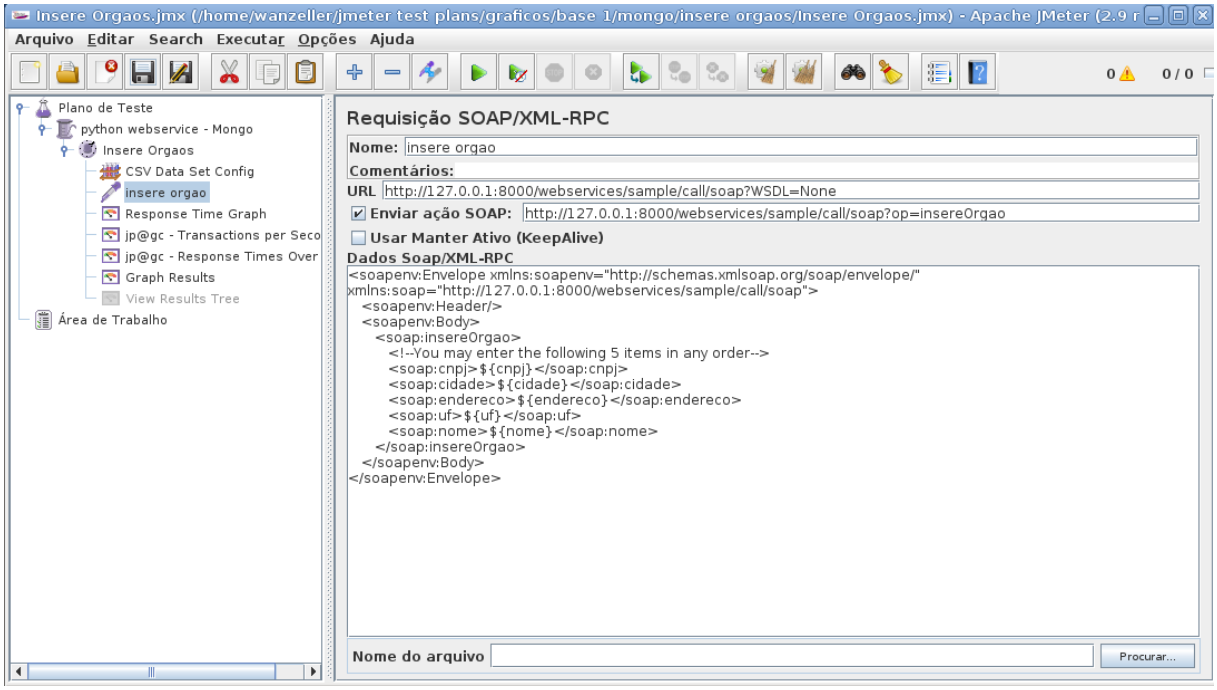

Figura 2.13: Testador de Requisição SOAP/XML - RPC

Os ouvintes nos permite ter acesso às informações geradas pelo JMeter durante os testes. Temos ouvintes que geram gráficos, gravam informações em arquivos, listam o retorno das requisições e outros vários. A seguir veremos o gráfico de resultados e o gráfico de tempo de resposta.

Gráfico de Resultados

O gráfico de resultados gera um gráfico com os tempos de todas as requisições. Na legenda do gráfico temos o tempo da requisição atual (preto), a média atual de todas as requisições (azul), a derivação atual (vermelho), e a vazão atual (verde), todas em milisecundos. A vazão representa o número de transações por minuto (os atrazos causados pelo processamento interno do JMeter não são considerados).

Gráfico de Tempo de Resposta

O gráfico de tempo de resposta plota uma linha no gráfico que descreve a evolução do tempo de resposta de cada requisição durante o teste.

Os elementos de configuração podem ser utilizadoos para configurar padrões e variáveis que serão utilizadas pelos testadores.

Elemento para Configuração de Dados CSV

Esse elemento de configuração é usado para ler linhas de um arquivo e armazená-las em variáveis.Podemos ver um exemplo na figura 2.14. Para realizar os testes de inserção de dados no banco esse elemento será de grande importância.

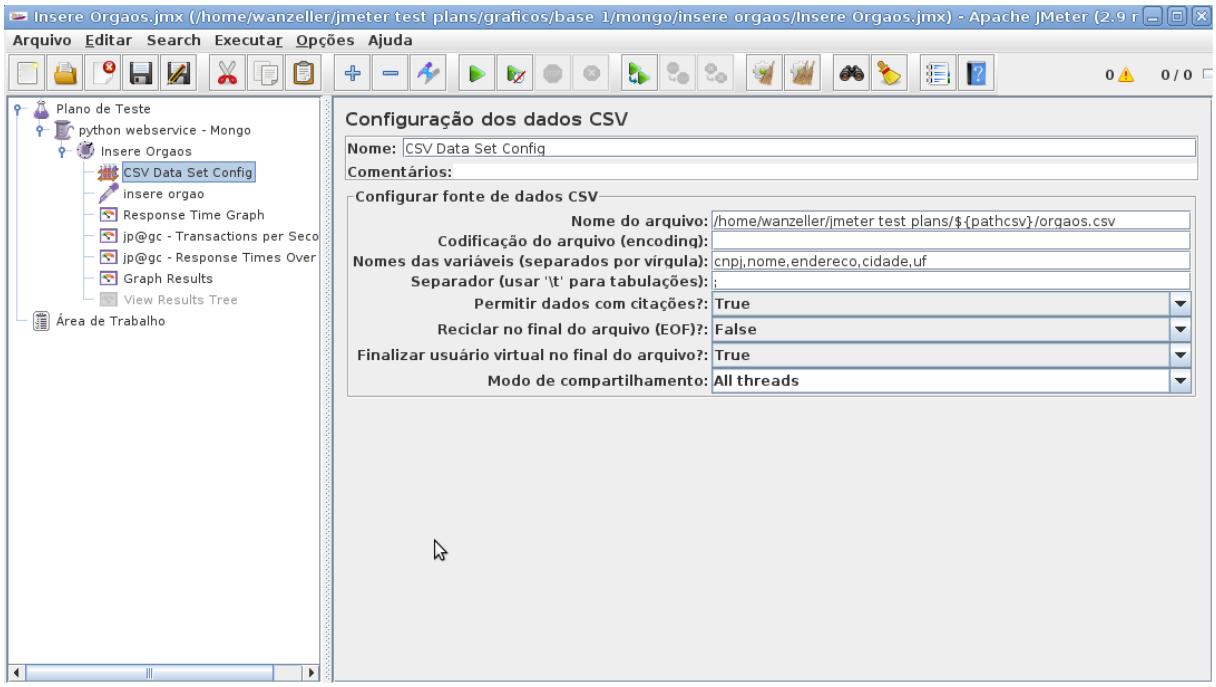

Figura 2.14: Elemento de Configuração de Dados CSV

Tabela 2.3: Declarações SQL vs Declarações MongoDB. Adaptado de [7]

SQL	MongoDB
CREATE TABLE users (id MEDIUMINT **NOT NULL** AUTO_INCREMENT, user_id **Varchar**(30), age **Number**, status **char**(1), **PRIMARY KEY** (id))	O documento é criado na primeira operação de inserção. Se o campo id não for especificado ele é automaticamente gerado. db.users.insert({ user_id: "abc123", age: 55, status: "A" })
ALTER TABLE users **ADD** join_date DATETIME	O MongoDB não amarra a estrutura das coleções. Não existe alteração estrutural no nivel das coleções. As alterações ocorrem no nível dos documentos. db.users.update({ }, { $set: { join_date: **new** Date() } }, { multi: **true** })
ALTER TABLE users **DROP COLUMN** join_date	db.users.update({ }, { $unset: { join_date: "" } }, { multi: **true** })
DROP TABLE users	db.users.drop()

Tabela 2.4: SQL Select vs MongoDB Select. Adaptado de [7]

SQL	MongoDB
`SELECT * FROM users`	`db.users.find()`
`SELECT id, user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1 } )`
`SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } )`
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`
`SELECT COUNT(*) FROM users`	`db.users.count()` ou `db.users.find().count()`
`EXPLAIN SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } ).explain()`

Tabela 2.5: Tipos de teste e sua característica de qualidade correspondente

Tipos de Teste	**Características de qualidade**
Funcionalidade	Funcionalidade
Interfaces	Conectividade
Carga	Continuidade, Performance
Produção	Operabilidade
Recuperação	Recuperação
Regressão id	Todas
Segurança	Segurança

Capítulo 3

Protótipo AFD e Plano de Testes

Esse capítulo apresenta a arquitetura e a implementação de um protótipo para o AFD (Assentamento Funcional Digital) que objetiva servir para a investigação proposta nesse trabalho e descrita na introdução. Como o protótipo arquitetural foi baseado em uma arquitetura orientada a serviços [22] que disponibiliza algumas das principais capacidades para manter os dados do AFD, iniciaremos o capítulo com a descrição de *Web service*. Logo em seguida será descrita as principais características do projeto como tecnologias utilizadas e arquitetura proposta. Para finalizarmos o capítulo descrevemos os planos de testes que foram desenvolvidos para os testes de performance.

3.1 Web-Service

Nessa seção daremos uma visão de o que é *web service*, e mostraremos um pouco sobre os seus principais componentes. Essa seção foi baseado no w3c schools [12] exeto quando citado explicitamente.

3.1.1 O que é web-service?

Web service são componentes que podem ser acessados via protocolo http. Atualmente é muito usado na comunicação entre aplicações diferentes. O acesso a um *web service* é via http, mas internamente existe dados formatados em xml que estão empacotados no protocolo SOAP (Simple Object Access Protocol).

Hoje várias aplicações podem acessar a web usando os *browsers* e nem sempre essas aplicações conversam entre si. Para que a comunicação entre essas diversar aplicações se tornasse possível, independente da plataforma em que estivessem desenvolvidas, foi criado o conceito de *web service*. Usando *web services*, as aplicações podem publicar suas funções para toda a web. Usando o XML para codificar e SOAP para transportar os dados, os *web services* elevaram as aplicações web para outro nível.

3.1.2 Componentes de um Web-Service

Um *web service* é formado por três elementos: SOAP, WSDL e UDDI.

SOAP

O SOAP (Simple Object Access Protoco) é um protocolo leve para troca de informações que foi criado pela Microsoft, Ariba e IBM para padronizar a transferência de dados em diversas aplicações, por isso, se dá em XML. Parte da sua especificação é composta por um conjunto de regras de como utilizar o XML para representar os dados. Outra parte define o formato de mensagens, convenções para representar as chamadas de procedimento remoto (RPCs) utilizando o SOAP, e associações ao protocolo HTTP.

SOAP é:

- Um protocolo de comunição;
- É usado para a comunicação entre aplicações;
- É um padrão para envio de mensagens;
- Sua comunicação se dá na internet;
- É independente de plataforma;
- É independente de linguagem de programação;
- É baseado em XML;
- Permite passar por *firewalls*;
- É uma recomendação do W3C;

Atualmente as aplicações se comunicam via RPC (Remote Procedure Calls), mas o HTTP não foi desenhado para isso. RPC possui problemas de compatibilidade e segurança; *firewalls* e servidores de *proxy* normalmente bloqueiam mensagens desse tipo. Para resolver esses problemas foi criado o protocolo SOAP.

Uma mensagem SOAP (Exemplo 3.1) é basicamente um documento XML que contem os seguintes elementos:

- Um elemento 'Envelope' que identifica o documento XML como uma mensagem SOAP;
- Um elemento 'header' que contem informações de cabeçalho;
- Um elemento 'body' que contem informações de chamadas e retornos;
- Um elemento 'Fault' que contem informações sobre erros e status;

Listing 3.1: Estrutura de uma mensagem SOAP

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
```

```
...
</soap:Header>

<soap:Body>
...
        <soap:Fault>
          ...
        </soap:Fault>
</soap:Body>

</soap:Envelope>
```

3.1.3 WSDL

WSDL é uma linguagem baseada em XML para localizar e descrever *web services*.

O WSDL (*Web Services Description Language*) é uma linguagem baseada em XML, com a finalidade de documentar as mensagens que o Web service aceita e gera (Exemplo 3.2). É uma espécie de contrato. Esse mecanismo padrão facilita a interpretação dos 'contratos' pelos desenvolvedores e ferramentas de desenvolvimento.

WSDL é:

- A linguagem padrão para descrever *web services*;
- É baseado em XML;
- É usado para localizar *web services*;
- É um padrão W3C.

Um WSDL descreve um *web service* usando pricipalmente os seguintes elementos:

Tabela 3.1: Elementos de um documento WSDL

Elemento	Descrição
<types>	Um container para a definição dos tipos de dados usados pelo *web service*
<message>	Definição dos dados que serão usados na comunicação
<portType>	Um conjunto de operações suportadas por um ou mais *endpoints*
<binding>	Um protocolo e especificação de dados para um *port type* específico

Abaixo temos uma fração simplificada de um documento WSDL:

Listing 3.2: WSDL

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>
```

```
<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

3.1.4 UDDI

UDDI é um serviço de diretório que permite às empresas descobrir, registrar e procurar *web services*. É baseado em padrões do W3C (*World Wide Web Consortium*) e IETF (*Internet Task Force*) como XML, HTTP e DNS.

Os benefícios de se usar UDDI são muitos. Antes do UDDI não havia padrão para as empresas divulgarem seus produtos e serviços para os seus consumidores e parceiros. Com o UDDI, por exemplo, se for definido um padrão para serviços de empresas aéreas, quando as empresas publicarem os seus serviços em um diretório UDDI as agências de viagem poderão procurar por esses serviços e iniciar imediatamente a comunicação.

3.2 O Protótipo

Para a execução dos testes de performance foi desenvolvido um Web service com as principais operações necessárias para manter os dados do AFD. O objetivo central da aplicação é manter os documentos da pasta funcional dos servidores. Como a aplicação deve armazenar aquivos, escolhemos gravar o arquivo no sistema operacional e armazenar o caminho para ele na base de dados. Na Figura 3.1 temos o modelo de casos de uso e na figura 3.2 temos uma breve descrição dos métodos do Web service. O objetivo ao se escolher realizar os testes de performance via *web service* foi o de flexibilizar ao máximo as implementações em diversos bancos de dados. Na tabela 3.2 a descrição das funcionalidades implementadas.

3.3 A Arquitetura do Projeto

Para cumprirmos o nosso objetivo, que é testar a nossa aplicação com diferentes bancos de dados, montamos uma arquitetura simples, mas que nos permitisse trocar as implementações da camada de persistência sem maiores esforços. Para a execução dos testes utilizamos o JMeter. A aplicação foi desenvolvida em Python com o apoio do framework web2py na implementação do *web service*. A linguagem de programação Python foi escolhida pela facilidade de encontrar drivers de diversos bancos de dados relacionais e não relacionais, além de ser uma linguagem orientada a objetos e de ampla utilização. O framework web2py foi adicionado ao projeto pelo motivo de suportar a implementação de

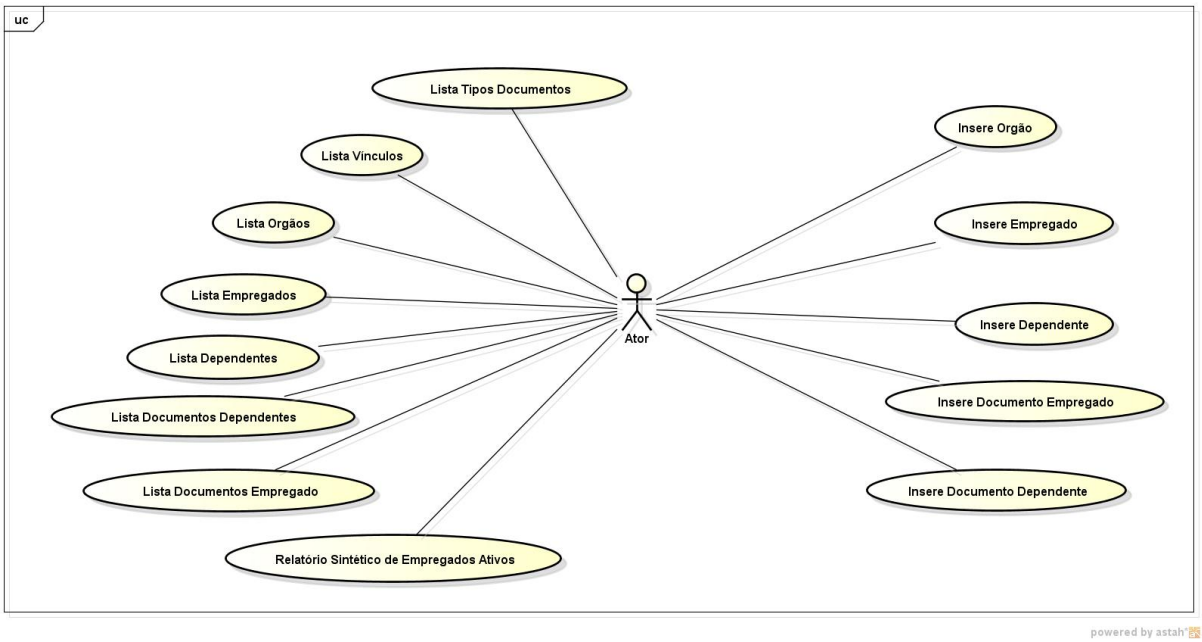

Figura 3.1: Modelo de Casos de Uso

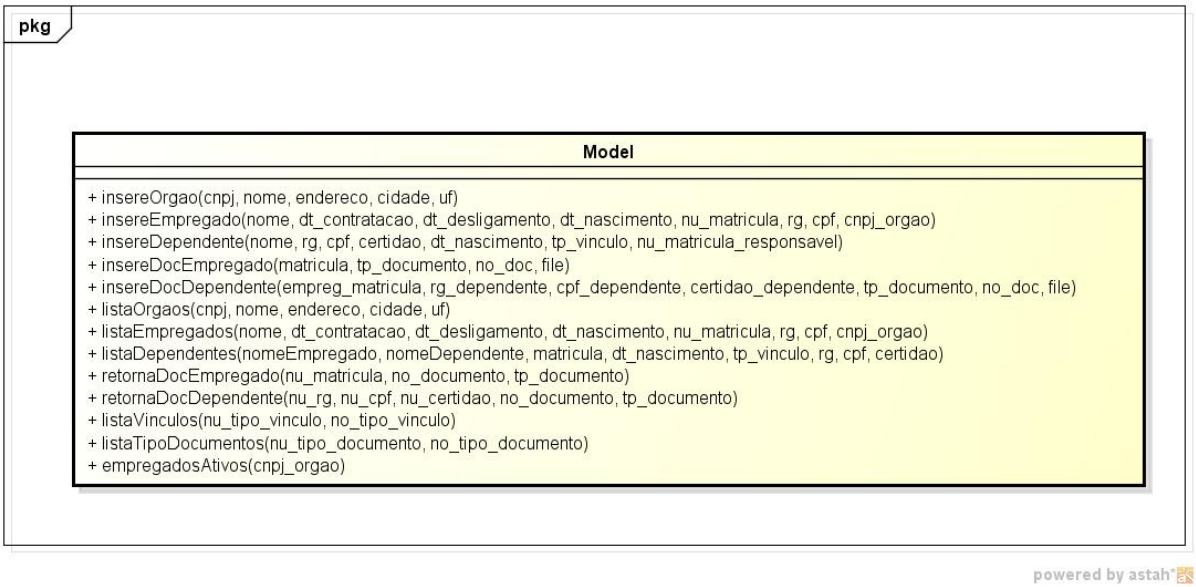

Figura 3.2: Descrição dos Métodos

Tabela 3.2: Descrição das Funcionalidades

Funcionalidade	**Descrição**
Insere Orgão	Permite inserir as unidades pagadoras ou orgãos que terão os dados dos empregados mantidos no sistema.
Insere Empregado	Permite inserir os empregados de cada orgão.
Insere Dependente	Permite inserir os dependentes de cada empregado.
Insere Documento Dependente	Permite inserir os documentos dos dependentes que farão parte da pasta funcional do empregado.
Insere Documento Empregado	Permite inserir os documentos que farão parte da pasta funcional do empregado.
Lista Orgãos	Lista os dados dos órgãos cadastrados no sistema.
Lista Empregados	Lista os dados dos empregados cadastrados no sistema.
Lista Dependentes	Lista os dados dos dependentes dos empregados.
Lista Documentos Empregados	Retorna os documentos da pasta funcional do empregado.
Lista Documentos Dependentes	Retorna os documentos dos dependentes dos empregados.
Relatório de Empregados Ativos	Calcula e exibe a quantidade de empregados ativos por orgão.
Lista Vínculos	Lista os valores possíveis para os tipos de vínculos entre empregados e dependentes.
Lista Tipo Documentos	Lista os valores possíveis para os tipos de documentos.
Remove Dependente	Exclui o depente e seus arquivos.
Desliga Empregado	Atualiza o status do empregado incluindo a data de desligamento.

web services de modo rápido e fácil, além da geração automática do WSDL (arquivo que contém a descrição das operações do Web service). A diagramação da arquitetura pode ser vista na figura 3.3

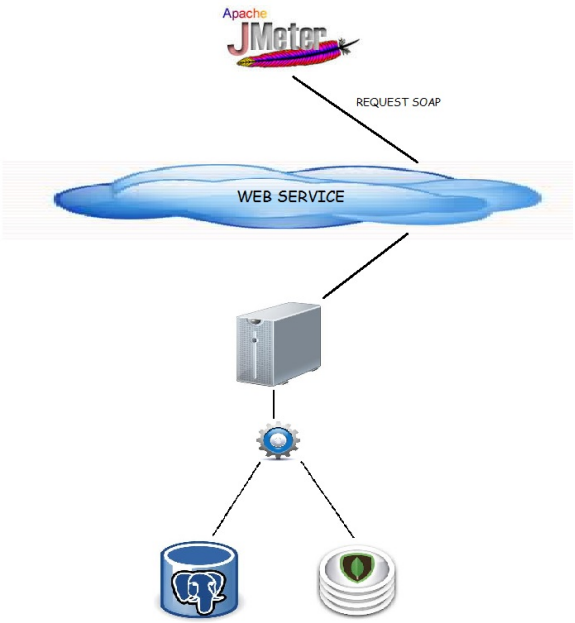

Figura 3.3: Arquitetura de Testes

3.3.1 web2py

Web2py é um *framework* para desenvolvimento ágil de aplicações web, software livre e gratuito. Ele é escrito e programável em Python. web2py foi inspirado pelo Ruby on Rails e Django. Tem seu foco no desenvolvimento ágil e segue o MVC (*Model View Controller*). Toda aplicação web2py é composta por *Models* (arquivos que contem a descrição dos dados), *Views* (arquivos que contem a descrição dos dados que serão apresentados), *Controlers* (arquivos que contem a lógica e *workflow* do negócio), *Cron Jobs* (tarefas que precisam ser executadas regularmente) e *Static Files* (imagens, *scripts*, folhas de estilos, etc.) [13].

Quando se trata de Web services, web2py oferece suporte para diversos protocolos, incluindo XML, JSON, RSS,CSV,XMLRPC,JSONRPC,AMFRPC, e SOAP. O web2py inclui um cliente e servidor SOAP (pysimplesoap) criado por Mariano Reingart. Uma facilidade encontrada é a geração automática do WSDL e da página com a descrição das capacidades [13].

3.4 Os Planos de Teste

Foi desenvolvido um plano de testes no JMeter para a maioria das funcionalidade da nossa aplicação. Os testes serão individualmente discutidos na seção de resultados do pró-

Tabela 3.3: Principais configurações dos Planos de Teste

Parâmetro	**Descrição**
Quantidade de usuários virtuais (threads)	Quanto maior o número de usuários virtuais, maior será o número de requisições simultâneas que a nossa aplicação terá que responder.
Tempo de inicialização dos usuários virtuais	Indica o tempo total para a inicialização de todos os usuários virtuais. Para encontrarmos o tempo entre a inicialização de cada usuário devemos dividir pelo total de usuários virtuais.
O local dos arquivos CSV	Esses arquivos devem ser gerados antes do início dos testes com o apóio de um script.
Intervalo de medição dos gráficos	É o intervalo de tempo em que o JMeter faz as medidas para plotar cada gráfico.

ximo capítulo. O testador utilizado no nosso projeto foi o de Requisição SOAP/XML - RPC. É nele que configuramos as requisições que serão feitas ao Web service da aplicação. Além de configurar uma requisição para cada plano de teste, temos como parametrizar outras configurações como a quantidade de usuários virtuais e o intervalo entre a inicialização de cada usuário. Na tabela 3.3 temos as principais configurações dos nossos planos de teste. É importante realçar que todas as capacidades da nossa aplicação foram desenvolvidas para validarem os dados da mesma forma, ou seja, as regras de negócio verificadas, tanto usando o mongodb quanto o postgresql, são as mesmas. O passo a passo da execução dos planos de testes se dividem em dois tipos: Inserção/Exclusão/Atualização e Consulta.

3.4.1 Planos de Teste de Inserção/Exclusão/Atualização

Os planos de Testes de Inserção executam os seguintes passos:

1. Configuração de Dados CSV - É indicado onde está o arquivo csv de onde as threads lerão os valores a serem enviados na requisição soap;

2. Requisição SOAP/XML-RPC - É configurada a URL do Web service e a requisição que será realizada. Cada requisição será montada com os dados lidos do arquivo csv. Cada thread lê uma linha diferente do arquivo.

3. Gráfico de Tempo de Resposta - Elemento responsável por gerar um gráfico a partir dos dados da requisição feita. O gráfico exibe a evolução do tempo de resposta das requisições feitas.

4. Gráfico de Resultados - Elemento responsável por exibir a evolução dos tempos das requisições, a média dos tempos das requisições, a derivação do tempo das requisições e a vazão.

As funcionalidades que seguem esses passos são: insere órgão, insere empregado, insere dependente, insere documento dependente, insere documento empregado, desliga empregado e remove dependente.

Os testes que manipulam os arquivos da pasta funcional, devido à limitação da arquitetura disponível, e o teste de inserção de órgãos, devido a baixa massa de dados, só foram realizados para 10 e 100 usuários simultâneos. O restante foi executado para 10, 100 e 500 usuários simultâneos

3.4.2 Planos de Teste de Consulta

Os planos de Testes de Consulta executam os seguintes passos:

1. Requisição SOAP/XML-RPC - É configurada a URL do Web service e a requisição que será realizada. Cada requisição será montada com os dados lidos do arquivo csv. Cada thread lê uma linha diferente do arquivo.

2. Gráfico de Tempo de Resposta - Elemento responsável por gerar um gráfico a partir dos dados da requisição feita. O gráfico exibe a evolução do tempo de resposta das requisições feitas.

3. Gráfico de Resultados - Elemento responsável por exibir a evolução dos tempos das requisições, a média dos tempos das requisições, a derivação do tempo das requisições e a vazão.

As funcionalidades que seguem esses passos são: lista órgãos, lista empregados, lista dependentes, lista documentos dos empregados, lista documentos dos dependentes e relatório de empregados ativos.

Cada plano de teste de consulta foi executado durante um minuto. Os testes que manipulam os arquivos da pasta funcional, devido à limitação da arquitetura disponível, só foram realizados para 10 e 100 usuários simultâneos. O restante foi executado para 10,100 e 500 usuários simultâneos.

Capítulo 4

Execução dos Testes e Análise dos Resultados

Nesse capítulo vamos descrever o ambiente onde os testes foram realizados, as métricas que foram escolhidas para medir a performance e os resultados obtidos.

4.1 Ambiente de testes

Os testes foram realizados em uma máquina física com as seguintes configurações:

- Sistema Operacional: Debian GNU/Linux 6.0
- Processador: Intel Pentium Quad Core
- Quantidade de Memória RAM: 4 GB
- MongoDB: Versão 2.4.0 padrão
- PostgreSQL: Versão 8.4.16 padrão
- Driver Python MongoDB: pymongo
- Driver Python PostgreSQL: psycopg

4.2 Massa de Dados

Conforme Molyneaux diz [30], a importância de prover a quantidade de dados de qualidade para um teste não pode ser exagerada. Segundo ele, a quantidade e a qualidade dos dados podem definir o sucesso ou insucesso dos testes. Para o nosso projeto foi desenvolvido um script em python para a geração da massa de dados. Os dados podem tanto ser inseridos diretamente na base de dados quanto em arquivos CSV, os quais serão utilizados durante os testes. Os arquivos (documentos para simular a pasta funcional) utilizados nos testes possuem tamanho médio de 400 KB. A não ser pela diferença das chaves primárias geradas nos dois bancos, os dados inseridos no MongoDB e no PostgreSQL são iguais. A quantidade de dados gerados também pode ser configurada pelo seguinte:

1. Quantidade de Unidades Pagadoras (Orgãos);

Tabela 4.1: Massa de Dados Utilizada

Quantidade de Unidades Pagadoras	**Descrição**
Quantidade de empregados por orgão	10
Quantidade de dependentes por empregado	100
Quantidade de documentos por empregado	5
Quantidade de documentos por dependente	2
Quantidade de dependentes excluídos	250
Quantidade de empregados desligados	250

2. Quantidade de empregados por orgão;
3. Quantidade de dependentes por empregado;
4. Quantidade de documentos por empregado;
5. Quantidade de documentos por dependente;

A quantidade de dados utilizados pode ser vista na tabela 4.1

4.3 Métricas

Quando se quer balancear o custo e a performance, todos os envolvidos na produção do software se preocupam com a execução de testes de performance. A avaliação de performance é necessária em todas as etapas do ciclo de vida de software e é requerida sempre que o arquiteto precisa comparar alternativasv [27]. Em um teste de performance a escolha das métricas é de grande importância. Segundo Raj Jain [27], escolher as métricas erradas é um dos erros mais comuns. Nesse trabalho a performance será avaliada pelo tempo médio de resposta.

4.4 Resultados

Como podemos ver nos gráficos apresentados a seguir, a performance dos dois bancos foram bastante próximas. Após executar os testes, foram salvos os tempos de todas as requisições feitas à aplicação e calculado o tempo médio de resposta, em milisecundos, para cada teste realizado.

Para os testes de consulta, exceto no teste 'Consulta Documentos do Dependente' o MongoDB sempre foi mais lento em relação ao PostgreSQL.

Na inserção de orgãos os tempos foram muito próximos e, como a quantidade de registros inseridos foram poucos, podemos dizer que a performance dos dois bancos foram iguais.

Ao inserir os dados dos empregados e dos dependentes, trabalhamos com uma quantidade de registros maior e assim podemos ver que o MongoDB foi sempre mais lento em relação ao PostgreSQL. Essa diferença de performance foi aumentando a medida em que a quantidade de usuários simultâneos foi incrementada.

No teste 'Consulta Usuários Ativos' é feito um cálculo interno e também foi utilizado agrupador nas consultas realizadas nos dois bancos. Mais uma vez o PostgreSQL foi mais rápido ao responder as requisições.

Ao testarmos a inserção de documentos, tanto de empregados quanto dos seus dependentes,percebemos que o tempo de resposta aumentou muito em relação aos outros teste. Isso se deve ao fato de que foram adicionalmente necessárias operações de criação e leitura de arquivos e diretórios. Esses testes foram realizados apenas para 10 e 100 usuários simultâneos, pois a arquitetura não nos permitiu mais. Ao inserir os documentos dos empregados podemos ver que a diferença se mostra maior ao testarmos com 100 usuários simultâneos, quando o PostgreSQL é aproximadamente um segundo mais rápido que o MongoDB. Já na inserção dos documentos dos dependentes, a performance dos dois bancos são bem parecidadas, com uma pequena vantagem para o MongoDB.

Ao testarmos a atualização de registros no teste 'Desliga Empregado' e a exclusão de registros no teste 'Remove Dependente', mesmo com operação de *join* nesse, também foi verificado que o PostgreSQL é mais rápido ao responder às requisições. É importante realçar que no teste 'Remove Depen'

Ao final dos testes pode-se verificar que nenhum bando de dados foi consideravelmente mais veloz que o outro e que, para praticamente todos os testes realizados, o PostgreSQL se mostrou mais rápido. Dessa maneira, para o cenário de manutenção dos dados do AFD, com a arquitetura, massa de dados e modelagem utilizadas, não é vantajoso utilizar o MongoDB para a persistência dos dados, visto que, mesmo com todos os recursos de segurança e controle de transações oferecidos pelo PostgreSQL, ele ainda continua sendo mais performático.

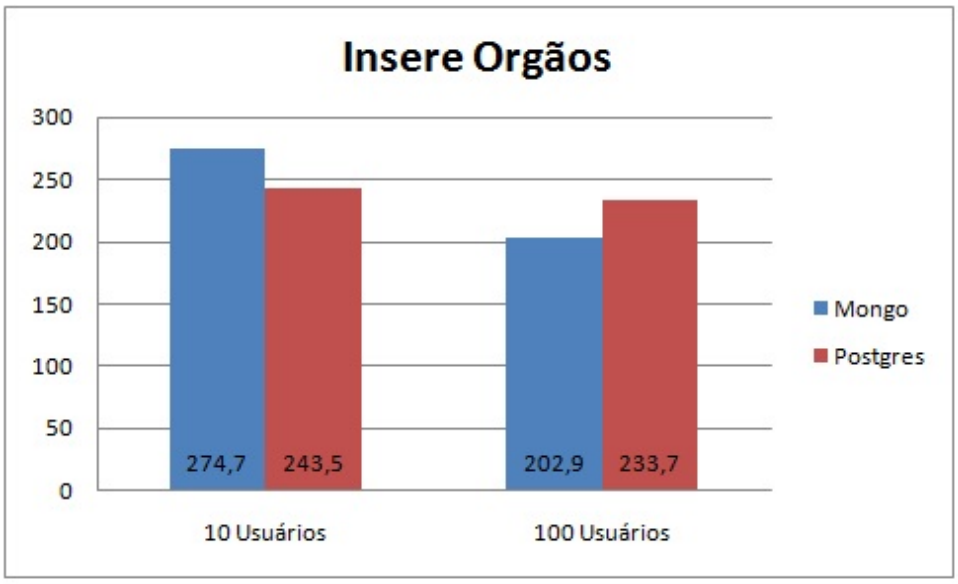

Figura 4.1: Resultados - Insere Órgãos

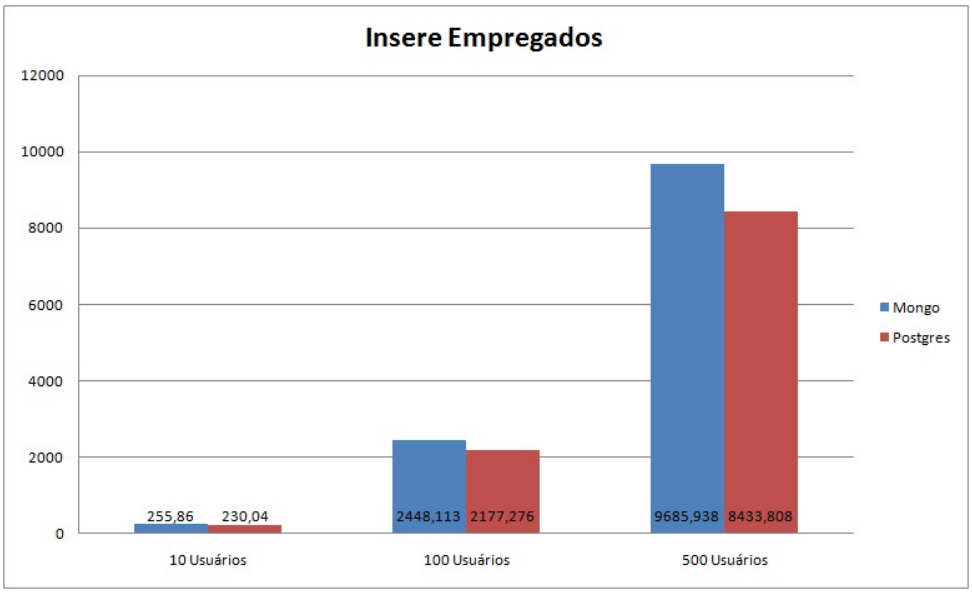

Figura 4.2: Resultados - Insere Empregados

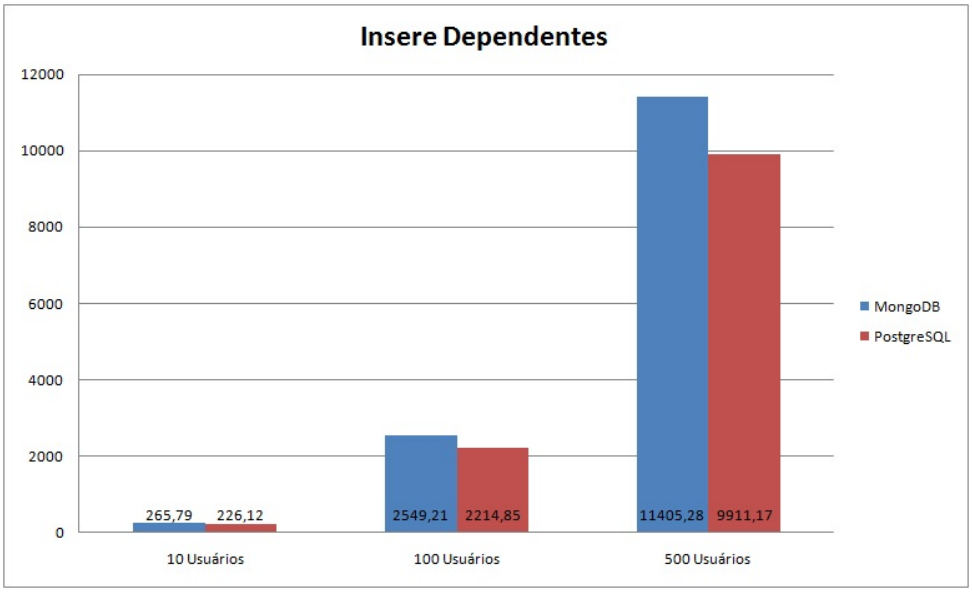

Figura 4.3: Resultados - Insere Dependentes

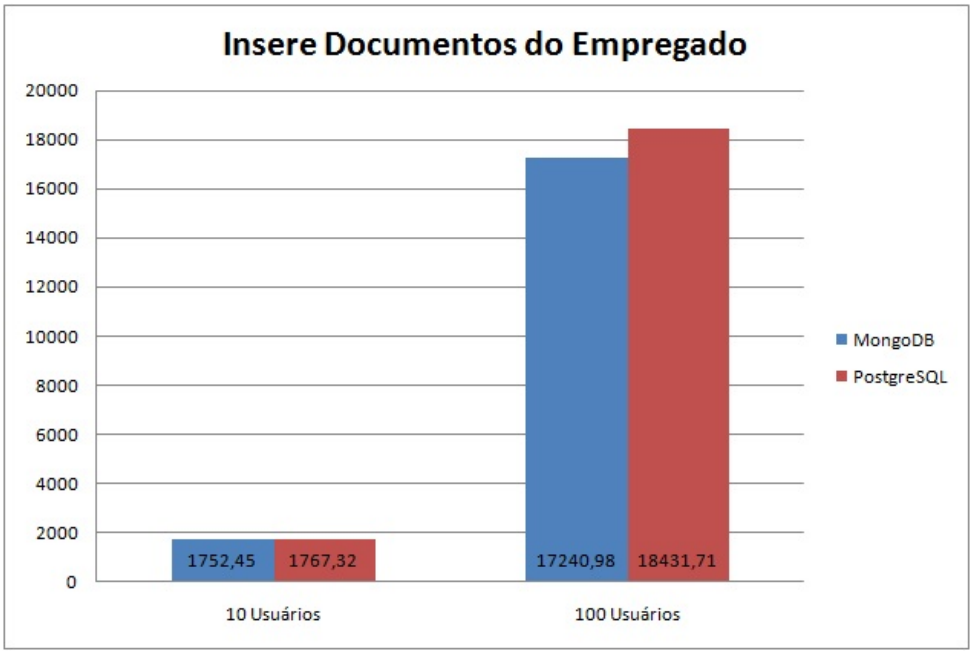

Figura 4.4: Resultados - Insere Documento do Empregado

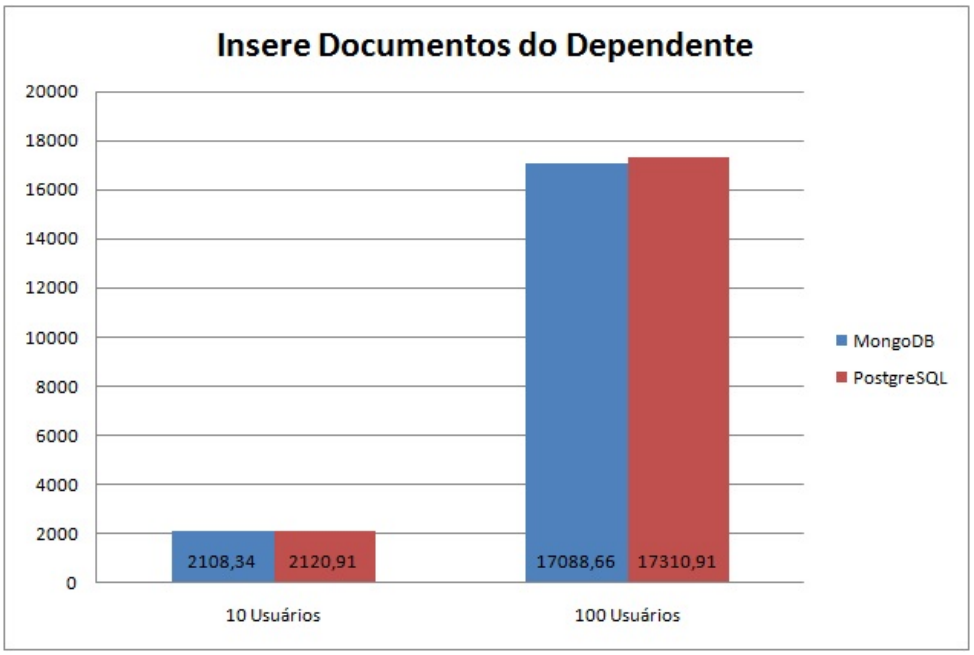

Figura 4.5: Resultados - Insere Documento do Dependente

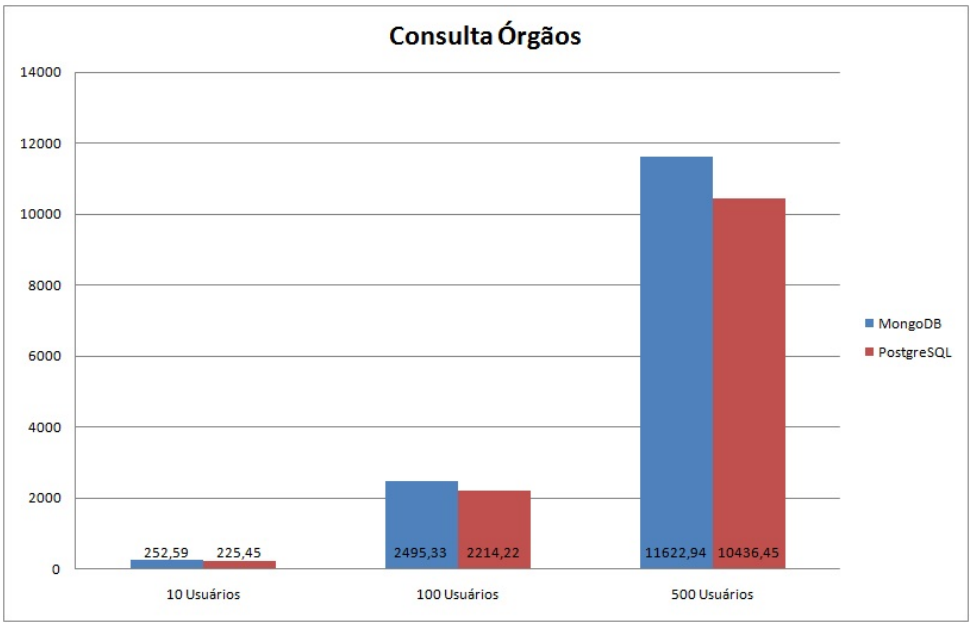

Figura 4.6: Resultados - Lista Órgãos

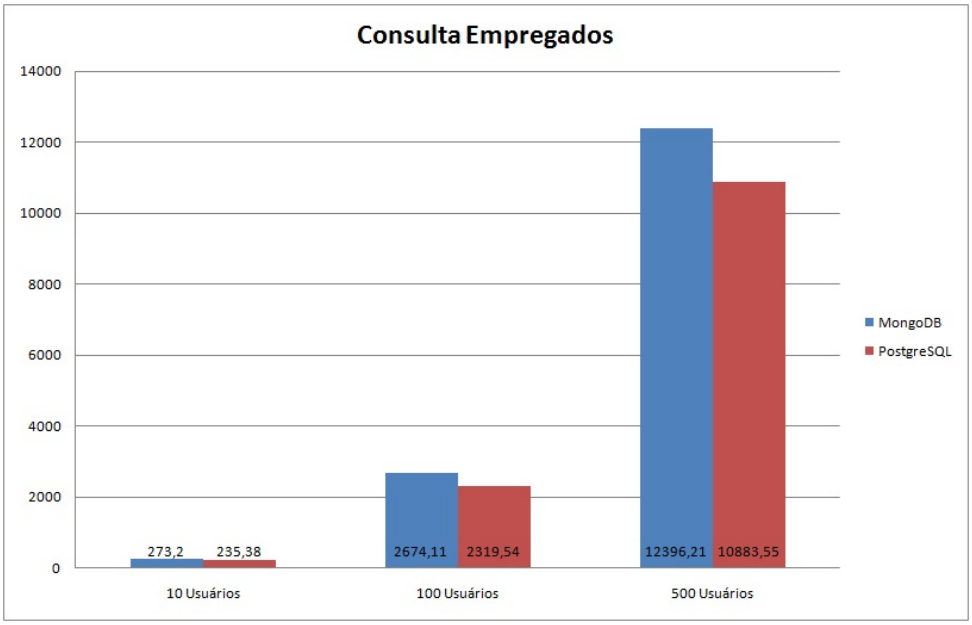

Figura 4.7: Resultados - Lista Empregados

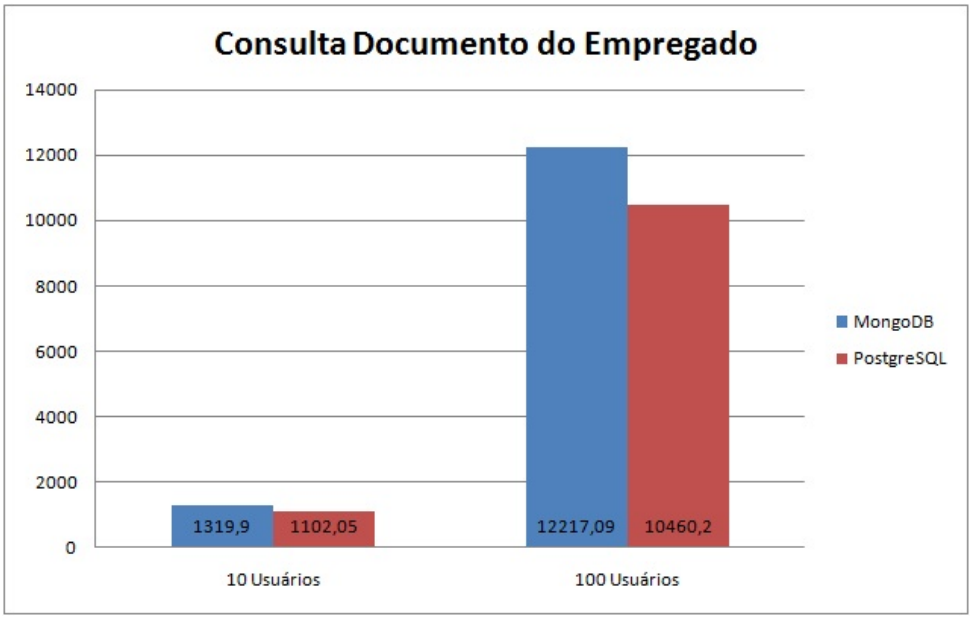

Figura 4.8: Resultados - Lista Documentos do Empregado

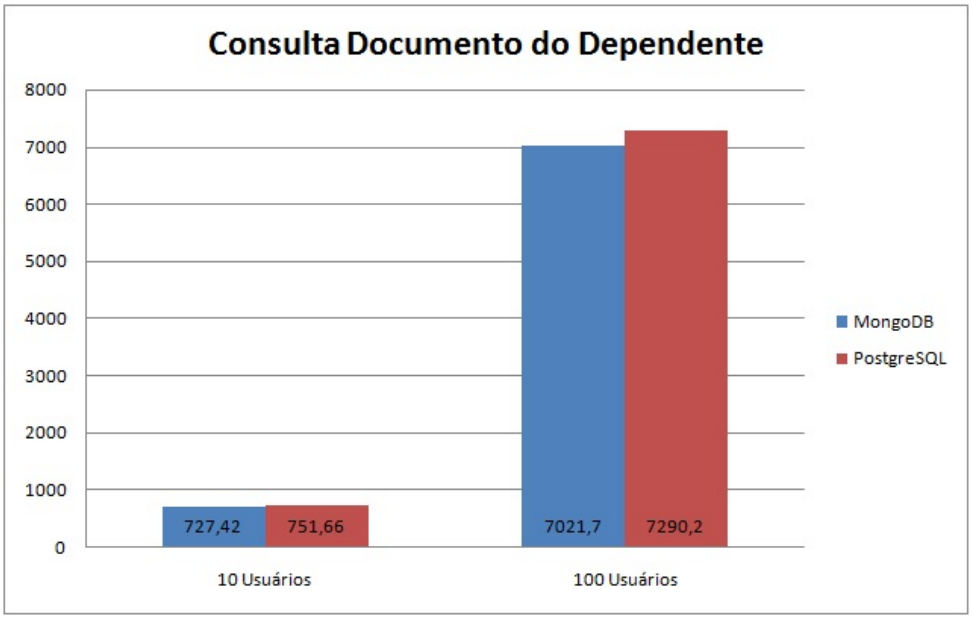

Figura 4.9: Resultados - Lista Documentos do Dependente

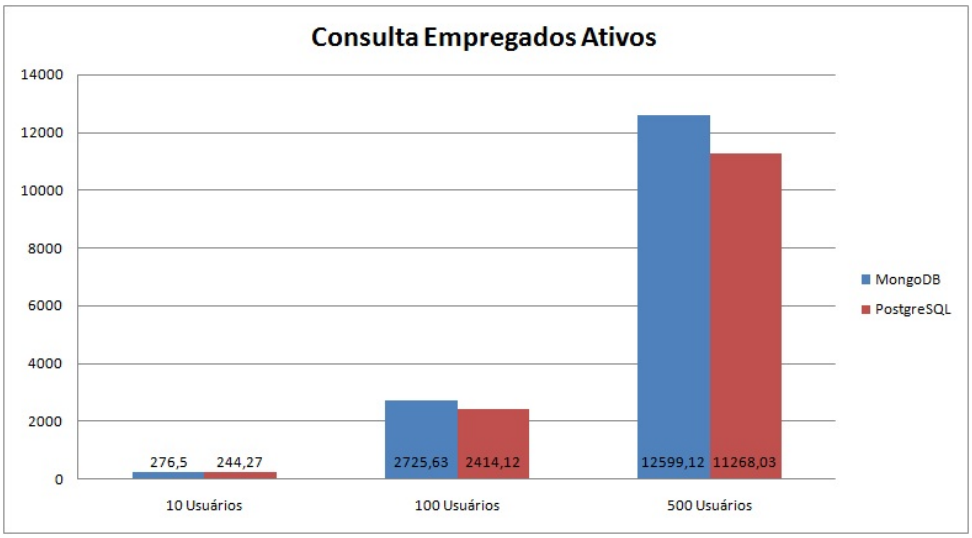

Figura 4.10: Resultados - Consulta Empregados Ativos

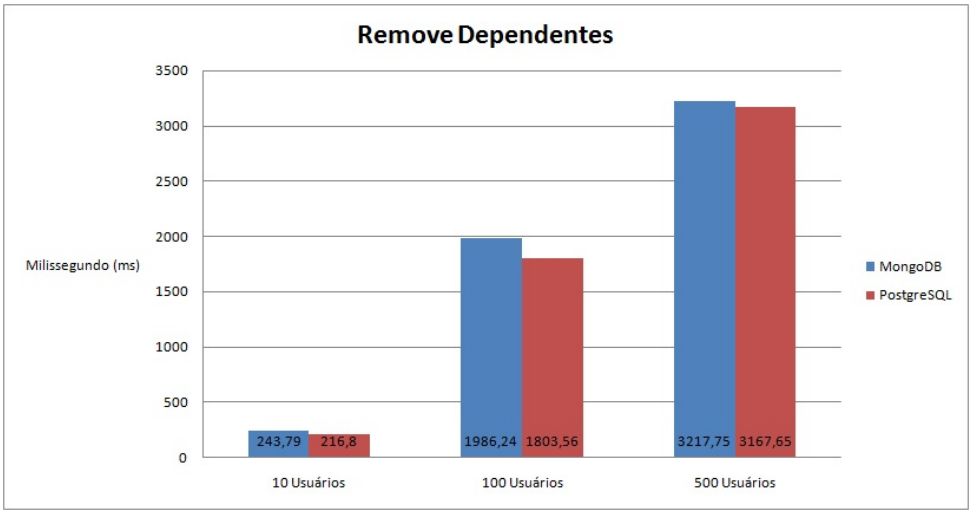

Figura 4.11: Resultados - Remove Dependentes

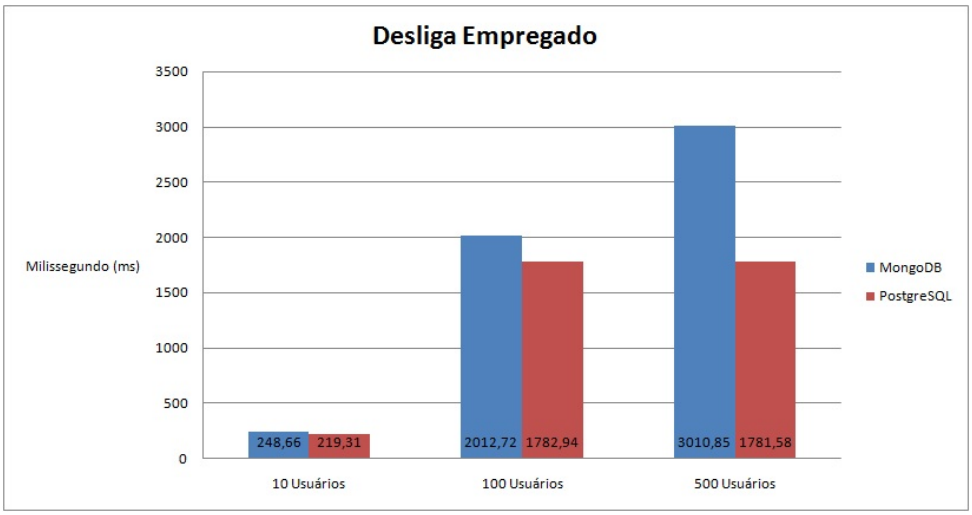

Figura 4.12: Resultados - Desliga Empregado

Capítulo 5

Considerações Finais e Projeto Futuros

Nesse capítulo temos algumas considerações finais sobre o projeto e também a enumeração de alguns possíveis trabalhos futuros.

5.1 Considerações Finais

É claro que o MongoDB já se figura entre os mais populares bancos de dados NoSQL e que o PostgreSQL já é uma solução consolidada. O nosso trabalho visou testar a performance dessas duas soluções em um cenário definido, mas ambos possuem aplicação nas mais diversas soluções feitas no mundo. Como exemplo temos a implantação do MongoDB no CartolaFC [6] da Globo.com e, em relação ao PostgreSQL, no site oficial [4] temos diversos casos de sucesso.

- O código fonte da aplicação, *scripts* utilizados e todos os planos de teste podem ser encontrados no repositório do projeto [1].

5.2 Projetos Futuros

Ao concluir esse trabalho é possível ver diversos outros testes para enriquecer os estudos sobres as tecnologias apresentadas nesse trabalho. A seguir vou enumerar três possíveis trabalhos:

1. Modelar a estrutura do MongoDB usando somente sub-documentos e verificar qual o impacto na performance;
2. Implementar a arquitetura em uma infraestrutura mais robusta, com mais de uma máquina executando o JMeter e realizando requisições distribuidas, além de realizar os testes com uma massa de dados maior. Na tabela 5.1, 5.2 e 5.3 temos uma sugestão de infra-estrutura para a realização de testes;
3. Expandir esses testes para outros bancos de dados como o MySQL e Cassandra.

Tabela 5.1: Infra-Estrutura para Trabalhos Futuros

SERVIDOR 1 - APLICAÇÃO - QTD: 1	
Sistema Operacional	Debian 6.0 - Squeeze com interface gráfica
SOFTWARES	
Python	Versão 2.6.6 com pymongo e psycopg2
PostgreSQL	8.4.16
MongoDB	2.4
web2py	2.5.1
Servidor Web Apache	2.2
HARDWARE	
Disco	30 TB
Memória	8 GB

Tabela 5.2: Infra-Estrutura para Trabalhos Futuros

JMETER MÁSTER - QTD: 1	
Sistema Operacional	Debian 6.0 - Squeeze com interface gráfica
SOFTWARES	
JMeter	2.9
HARDWARE	
Disco	200 GB
Memória	6 GB

Tabela 5.3: Infra-Estrutura para Trabalhos Futuros

JMETER - CLIENTES - QTD: 5	
Sistema Operacional	Debian 6.0 - Squeeze com interface gráfica
SOFTWARES	
JMeter	2.9
HARDWARE	
Disco	200 GB
Memória	4 GB

Referências

[1] 47

[2] Apache jmeter. http://jmeter.apache.org. Acessado em 02 de julho de 2013. vii, 22, 23

[3] Apresentação - assentamento funcional digital. www.sigepe.gov.br/assentamento-funcional-digital/apresentacao-assentamento-funcional-digital. Acessado em 23 de Abril de 2013. 2

[4] Casos de sucesso postgresql. http://www.postgresql.org/about/casestudies/. Acessado em 14 de novembro de 2013. 47

[5] Emc digital universe. http://www.emc.com/collateral/demos/microsites/emc-digital-universe-2011/index.htm. Acessado em 26 de Janeiro de 2013. 4

[6] Implantação do mongodb no cartolafc. http://www.gonow.com.br/blog/2011/07/29/o-mongodb-aplicado-ao-cartolafc-da-globo-com/. Acessado em 14 de dezembro de 2013. 47

[7] Mongodb official site. http://www.mongodb.org/. Acessado em 27 de Janeiro de 2013. vii, viii, 13, 15, 16, 17, 18, 26, 27

[8] nosql database org. http://nosql-database.org/. Acessado em 09 de Dezembro de 2012. 11, 12

[9] Recomendações para digitalização de documentos arquivísticos permanentes. http://www.emc.com/collateral/demos/microsites/emc-digital-universe-2011/index.htm. Acessado em 23 de Abril de 2013. 1

[10] Sigepe.org. www.sigepe.gov.br. Acessado em 23 de Abril de 2013. 2

[11] Testlink. http://testlink.sourceforge.net/. Acessado em 02 de julho de 2013. vii, 22

[12] W3c schools. http://www.w3schools.com/. Acessado em 25 de junho de 2013. 29

[13] web2py. http:///www.web2py.com. Acessado em 02 de novembro de 2013. 35

[14] K. Bakshi. Considerations for big data: Architecture and approach. In *Aerospace Conference, 2012 IEEE*, pages 1 –7, march 2012. 7

[15] D. Bollier, Communications, and Society Program (Aspen Institute). *The Promise and Peril of Big Data*. Aspen Institute, Communications and Society Program, 2010. 5

[16] Vinayak R. Borkar, Michael J. Carey, and Chen Li. Big data platforms: What's next? *XRDS*, 19(1):44–49, 2012. 1, 4

[17] BRAIL. Portaria normativa mp, n. 3, Novembro 2011. 1

[18] Ricardo W. Brito. Banco de dados NoSQL x SGBDs relacionais: Análise comparativa. 2010. 10, 12

[19] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. 8

[20] Ramez Elmasri and Shamkant B. Navathe. *Sistemas de Banco de dados*. Pearson Addison Wesley, 6 edition, 2011. 6

[21] Benjamin Erb. Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012. vii, 8

[22] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. 2, 29

[23] Gartner. Big data. http://www.gartner.com/it-glossary/big-data/. Acessado em 14 de Dezembro de 2012. 4

[24] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363 –366, oct. 2011. 12

[25] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336 –341, dec. 2011. 7, 10, 11, 12

[26] E. Hewitt. *Cassandra: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010. 7

[27] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991. 39

[28] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12 –14, feb. 2010. 6, 11

[29] Sam Madden. From databases to big data. *Internet Computing, IEEE*, 16(3):4 –6, may-june 2012. 4, 5

[30] Ian Molyneaux. *The Art of Application Performance Testing - Help for Programmers and Quality Assurance.* O'Reilly, 2009. 21, 38

[31] Kai Orend. Analysis and classification of nosql databases and evaluation of their ability to replace an object-relational persistence layer, 2010. vii, 6, 8, 9, 10, 14, 15, 16, 17

[32] E. Rios and T. MOREIRA. *Teste de software*. Alta Books, 2006. 19, 22

[33] Cezar Taurion. Big data = volume + variedade + velocidade. https://www.ibm.com/developerworks, 2012. Acessado em 14 de Dezembro de 2012. 4

[34] Cezar Taurion. Você realmente sabe o que é big data? https://www.ibm.com/developerworks, 2012. Acessado em 14 de Dezembro de 2012. 2

[35] S. Tiwari. *Professional NoSQL*. Wrox Programmer to Programmer. Wiley, 2011. 10, 13

[36] Guoxi Wang and Jianfeng Tang. The nosql principles and basic application of cassandra model. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 1332–1335, 2012. 8, 9