Universidade Federal de Pernambuco

Centro de Informática

Diogo Filipe Dornelas Falcão

Uma Arquitetura de *Cloud Computing* para análise de *Big Data* provenientes da *Internet Of Things*

Trabalho de Graduação

Recife, Fevereiro de 2014.

Universidade Federal de Pernambuco

Centro de Informática

Uma Arquitetura de *Cloud Computing* para análise de *Big Data* provenientes da *Internet Of Things*

Trabalho apresentado como requisito parcial para obtenção do grau de bacharel em Engenharia da computação pelo Centro de Informática da Universidade Federal de Pernambuco.

Aluno: Diogo Filipe Dornelas Falcão

Orientadora: Prof. Judith Kelner

Recife, Pernambuco, Brasil

28 de fevereiro de 2014.

Resumo

Os dados gerados pela *Internet Of Things* (Internet das Coisas) crescerão exponencialmente em consequência do aumento dos números de dispositivos conectados, chegando a alcançar Zetabytes (10^{21} bytes) de dados a partir de 2010 [1]. Novos modelos para armazenamento em *Cloud* (nuvem) e tratamento de grande volume de dados de forma distribuída também são de suma importância neste contexto.

Este trabalho de graduação tem como objetivo propor e desenvolver uma arquitetura de *Cloud*, a qual será capaz de processar dados provenientes de dispositivos conectados, na era da *Internet Of Things* e *Big Data*, com o intuito de gerar análises para facilitar a tomada de decisão. As decisões poderão ser por parte de sistemas computacionais, utilizando a *Cloud* como interface de controle, ou por seres humanos monitorando a *Cloud*.

Palavras-Chave — *Big Data, Sensing as a Service, Internet Of Things, Redes de Sensores de Grande Escala, Computação em Nuvem, Gerenciamento de Dados.*

Agradecimentos

Agradeço este trabalho imensamente aos meus pais que me apoiaram durante toda a graduação.

Aos meus amigos que participaram direta e indiretamente para a conclusão do mesmo.

E a professora Judith Kelner em conjunto com o GPRT pela oportunidade de crescimento, profissional e pessoal, ao longo deste projeto.

I. Indices

I. Indices 5
II. Introdução 8
III. Contextualização 9
A. *Internet Of Things* 9
1. Processamento 13
2. Armazenamento 15
B. *Cloud* Computing 16
IV. Banco de Dados 20
A. Tipos de Banco de Dados 20
Flat File Database 20
1. Relacional Database Management System – RDBMS 20
2. NoSQL Database 22
V. Titan 30
VI. Apache Cassandra 36
VII. TinkerPop Stack 38
A. *Rexster* 39
B. Pipes 40
C. Gremlin 41
VIII. Faunus 43
IX. Esquema da Arquitetura 44
X. Implementação 47
A. Avaliação 49
XI. Linguagem Específica do Domínio 56
XII. Conclusões e Trabalhos Futuros 61
XIII. Referências 62
XIV. Apêndice 66

INDICES DE TABELAS

TABELA 1 - BANCO DE DADOS - TAMANHOS (VICKNAIT, C. ET AL. 2010) 26
TABELA 2 - RESULTADO DO TEMPO DE CONSULTA EM MILISSEGUNDOS (VICKNAIT, C. ET AL. 2010) 26
TABELA 3 - COMPARATIVO ENTRE AS CATEGORIAS DE BANCO NOSQL [6] 29
TABELA 4- SUMÁRIO FLAT FILE, RBDMS E NOSQL [6] 30
TABELA 5 - COMPARAÇÃO ENTRE BACKENDS 34
TABELA 7 - PASSOS COMPOSTOS IMPLEMENTADOS 57
TABELA 8 - PASSOS NATIVOS DO GREMLIN 59
TABELA 9 - FUNÇÕES IMPLEMENTADAS 61

INDICES DE FIGURAS

FIGURA 1 - ESQUEMÁTICO DA INTERNET DAS COISAS EXIBINDO USUÁRIOS E APLICAÇÕES BASEADO EM ÁREAS DE BIG DATA[7] 12
FIGURA 2 - VISÃO GERAL CLOUD COMPUTING 18
FIGURA 3 - BLUEPRINTS STACK [24] 39
FIGURA 4 - ESQUEMÁTICO GREMLIN [24] 41
FIGURA 5 - ESQUEMA DA ARQUITETURA 45
FIGURA 6 - MODELAGEM EM GRAFO DOS DADOS 48
FIGURA 7 - HISTOGRAMA - CONSULTAS POR TEMPO DE EXECUÇÃO 50
FIGURA 8 - HISTOGRAMA - CONSULTAS POR TEMPO DE EXECUÇÃO 51
FIGURA 9 - HISTOGRAMA - CONSULTAS POR TEMPO DE EXECUÇÃO 51
FIGURA 10 - GRÁFICO - CONSULTAS POR TEMPO DE EXECUÇÃO 52
FIGURA 11 - GRÁFICO - CONSULTAS POR TEMPO DE EXECUÇÃO 53
FIGURA 12 - GRÁFICO - CONSULTAS POR TEMPO DE EXECUÇÃO 54
FIGURA 13 - GRÁFICO - CONSULTAS POR TEMPO DE EXECUÇÃO 54

II. Introdução

Com a redução no custo de produção de dispositivos embarcados e na expansão das redes sem fio, a quantidade de dispositivos do tipo sensores, atuadores, telefones móveis e computadores de modo geral têm crescido vertiginosamente. Tal concentração de fontes de processamento permite capturar, processar e analisar muitas vezes em tempo real, dados provenientes de bilhões de dispositivos conectados que servem a diferentes aplicações, desde monitoramento de ambientes, aplicações industriais e comerciais além de aplicações voltadas para melhorar a vida dos usuários.

Este desenvolvimento acarretou em que já no ano de 2008 a quantidade de dispositivos conectados a internet ultrapassou a quantidade de seres humanos conectados. Este fenômeno tende a crescer ainda mais nos próximos anos, pesquisas do *ABI Research* indicam que mais de 50 bilhões de dispositivos estarão conectados a internet em 2020, um número 6 vezes maior que a quantidade esperada de seres humanos conectados no mesmo período [2].

Em 1998 com o fenômeno de bilhões de dispositivos conectados foi introduzido o conceito da *Internet of Things (IoT)* ou Internet das coisas. Desde então diversas pesquisas e tecnologias vêm sendo desenvolvidas na área, protocolos como o IPV6, por exemplo, surgiram para prover endereçamento diante desta nova demanda. Além disto, novos modelos para o armazenamento e processamento de grande volume de dados de forma distribuída também são de suma importância neste contexto.

Em consequência disto, surge a necessidade de gerenciar e obter informações a partir de grande volume de dados, estruturados ou não, gerados frequentemente por dispositivos heterogêneos, caracterizando assim, um cenário conhecido como *Big Data*.

Portanto, fica clara a necessidade de uma estrutura de processamento e armazenamento que seja escalável horizontalmente. Tal como o *Hadoop* [3], que é capaz de processar um grande volume de dados, de forma distribuída, através de *clusters* utilizando o conceito de *Map-Reduce* [4]. Como também a utilização de bancos de dados *NoSQL* [5] que podem fazer uso de múltiplas instâncias, distribuídas em *clusters*, para prover balanceamento de carga.

III. Contextualização

A. Internet Of Things

O termo *Internet Of Things* foi inicialmente utilizado por Kevin Ashton em 1999, no contexto de gerenciamento de uma cadeia de suprimentos, entretanto este conceito foi expandido para diversas áreas. De acordo com a *Cisco Internet Business Solutions Group* (IBSG), o efeito *Internet Of Things* pode ser considerado como tendo início a partir do ponto em que a quantidade de dispositivos ou objetos conectados se tornou superior a quantidade de pessoas.

As grandes cidades podem atualmente serem vistas como um complexo agregado de dispositivos eletrônicos esteja eles conectados a internet ou não, estes dispositivos constituem uma infraestrutura básica que fazem parte de serviços essenciais como, por exemplo, na área da saúde com a integração de vários serviços de exames e análises da água potável que é consumida diariamente. Este é um exemplo importante, pois se estima que as cidades percam em torno de 50% de toda a água potável devido a uma infraestrutura obsoleta e ineficiente, este processo de perda poderia ser evitado com o uso eficiente de sensores inteligentes, capazes de identificar e localizar vazamentos assim que ocorram, evitando assim, o desperdício da água transportada e consequentemente redução dos custos.

Na área de transporte, sensores que fazem parte de um automóvel podem enviar sinais em tempo real para um algoritmo sofisticado em um processador no próprio veículo, que pode tomar decisões para aprimorar a segurança da locomoção, evitando colisões ou mau uso dos seus componentes, outras informações podem ser repassadas a uma central que monitore o percurso, gerenciando a forma como o usuário dirige o veículo, podendo gerar benefícios para o usuário, tais como: descontos em seguros, redução nas taxas de trânsito, entre outros. Sensores também podem enviar informações que mostram se o veículo está sendo furtado e, portanto tomar decisões como o bloqueio de sua condução e acionamento da força policial em tempo real, e de forma automática sem a intervenção do usuário.

Outros exemplos incluem a área de energia com o uso de medidores de consumo digitais, na área de comunicações com o constante crescimento de smartphones e dispositivos portáteis, na educação com a popularização de *tablets* e computadores portáteis, como acontece no estado de Pernambuco onde em algumas escolas públicas já são distribuídos estes dispositivos de forma gratuita para os alunos, além do aumento no uso de leitores de livros digitais os chamados e-books.

Sendo assim, com o desenvolvimento das cidades as soluções tradicionais estão atingindo seus limites e torna-se necessário explorar o potencial das tecnologias não só de eletrônicos em geral, mas principalmente de dispositivos embarcados, introduzindo então o conceito de cidades inteligentes.

Tornar uma cidade inteligente não é uma tarefa simples e rápida. Existem muitos casos reais de esforços neste sentido, no Rio de Janeiro um bom exemplo é a criação do Centro de Operações, assim como o exemplo de Pernambuco já citado, também vemos esforços em outras cidades [6].

Com tecnologias cada vez mais miniaturizadas e a popularização de dispositivos que utilizam a tecnologia RFID (*Radio Frequency IDentification*), além do crescimento de tecnologias sem fio como Bluetooth e serviços de telefonia como o 3G e o 4G-LTE, podemos colocar inteligência nos limites mais externos das redes a um baixo custo, não apenas as pessoas se comunicam, mas cada vez mais os objetos se comunicam uns com os outros criando uma verdadeira Internet das coisas.

A tendência é que seja criada uma relação simbiótica entre o mundo físico e o mundo digital, com entidades físicas tendo também sua única identidade digital, podendo com esta comunicar-se e interagir com outras entidades do mundo virtual. Sejam objetos ou pessoas que estão conectadas, existindo assim bilhões de objetos identificáveis e conectados com enormes *datas centers* em *nuvem*, funcionando de forma interoperacional com os outros dispositivos, permitindo assim que os objetos façam parte dos sistemas de informação. Podendo adicionar inteligência ao mundo físico que rodeia a nossa sociedade afim de que os processos de gestão e operação das cidades sejam

mais descentralizados, com decisões sendo tomadas localmente, melhorando o seu desempenho, escalabilidade e aumentando a rapidez das decisões.

Entretanto alguns desafios ainda são encontrados, muitas cidades ainda não oferecem o suporte necessário para sustentar o advento destas novas tecnologias, seja com uma rede de comunicação precária ou por causa do alto custo dos dispositivos, causando um possível retardo na adoção das tecnologias que fazem parte e são essenciais para o contexto de *Internet Of Things*-IoT.

Consequentemente é necessário um estudo mais profundo nessa área, devido à importância iminente dos dispositivos integrados no contexto geral de cidades inteligentes (*Smart Cities*).

No paradigma de *Internet Of Things* o resultado da geração desta grande quantidade de dispositivos, foi a geração de grandes volumes de dados que precisam ser processados de forma eficiente e de fácil interpretação, além de um processo de armazenamento capaz de lidar com o grande volume de dados heterogêneos que é gerado de forma distribuída.

Estes dados caracterizam claramente um fenômeno conhecido como *Big Data*, uma descrição mais completa encontra-se no tópico posterior.

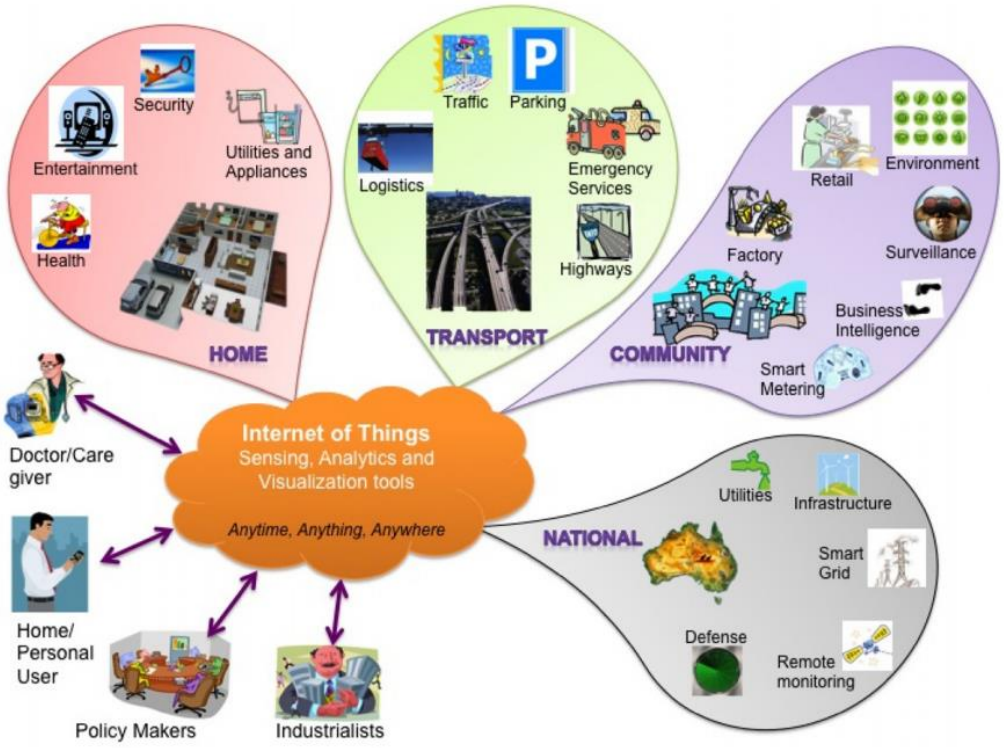

Figura 1 - *Esquemático da Internet das coisas exibindo usuários e aplicações baseado em áreas* de Big Data[7]

Dados vêm sendo coletados a uma velocidade nunca vista antes, gerados tanto pela crescente quantidade de dispositivos conectados (*Internet Of Things*) quanto por usuários, chegando até a dobrar de tamanho em um período de 18 meses. Estes dados têm uma característica em comum, são gerados em grande volume, com uma grande variedade e velocidade.

Os dados podem ser gerados através de análises de imagens de satélite, arquivos de *log*, vídeos, fotos, interações sociais dos mais diversos tipos, arquivos na web e até mesmo pesquisas e análises científicas.

Estima-se que, por exemplo, um motor de avião da Boeing funcionando durante 30 minutos produz em torno de 10 terabytes de dados, ou seja, um avião com 4 motores gera um total de 640 terabytes de dados, apenas em uma viagem atravessando o oceano Atlântico, com 25.000 vôos realizando rotas semelhantes em todo o mundo diariamente, a quantidade final de dados produzida alcança os valores de 16.000.000 terabytes ou 16 exabytes (10^{18} bytes). Outro exemplo, considerando o Twitter, contendo aproximadamente 200 milhões de usuários que fornecem em média 90 milhões de “tweets” diários,

gerando algo em torno de 8 terabytes de dados por dia, número de dados gerados 8 vezes maior que a quantidade de dados gerados pela bolsa de valores de Nova York em um dia de atividades.

Entretanto, os principais desafios de trabalhar com *Big Data* encontram-se principalmente em, como processar e armazenar estes dados, para que seja possível obter análises relevantes com alto desempenho de processamento, com o mínimo de perda de dados e da forma mais segura possível.

1. Processamento

Para processamento de *Big Data*, algumas propostas vêm sendo investigadas. Algumas destas pesquisas estão na área de *Stream Processing* [8], onde o paradigma de programação relacionado ao SIMD (Instruções únicas e múltiplos dados) é usado, o processamento explora o conceito de arquiteturas paralelas, principalmente as unidades de ponto flutuante presentes em GPUs de placas gráficas ou FPGAs (*Field-programmable gate array*), tecnologia muita usada para processamento de imagens e vídeos. Outra abordagem é o *Complex Event Processing* [9], baseado na análise de eventos simples sendo possível realizar inferências mais complexas sobre os dados. Por exemplo, se um grupo de usuários está realizando buscas por lojas de peças de roupas formais, e alguns destes usuários também pesquisam sobre presentes, é possível inferir que existe algum evento comum a todos nesta análise, provavelmente um casamento ou uma festa.

Entretanto, essas técnicas estão limitadas quanto à arquitetura de hardware (Paralelismo no caso do *Stream Processing*) ou tipos de dados (Strings ou relacionamentos no caso do *Complex Event Processing*), sendo necessária uma abordagem mais genérica a fim de lidar com o conceito de *Big Data*, obtendo o máximo possível dos dados heterogêneos sem depender da arquitetura do hardware.

O *Map-Reduce* é um modelo que foi proposto para o processamento de grandes conjuntos de dados, através da aplicação de tarefas independentes entre si. Programas em *Map-Reduce* são baseados em estilos de programação funcional, capazes de realizar processamento de grandes volumes de dados de

forma paralela. Composto por duas fases, esses programas se dividem na fase de *Map* onde ocorre sumarização dos dados, como por exemplo, uma contagem de uma determinada palavra presente em uma palavra menor, é nesta fase onde os elementos individuais são transformados e quebrados em pares do tipo Chave-Valor. Na fase de *Reduce*, a mesma recebe como entrada a saída da fase *Map*, a partir disto é realizado procedimentos de filtragem e ordenamento dos dados, que agrupam os pares semelhantes em conjuntos menores.

Tipicamente executados em *clusters*, o *Map-Reduce* permite uma massiva escalabilidade através de centenas ou milhares de máquinas distribuídas.

Diversas bibliotecas que utilizam o *Map-Reduce* foram escritas para vários tipos de linguagem de programação, com diferentes níveis de otimizações, entretanto a implementação mais popular do modelo utilizada é o *Apache Hadoop*.

a) Apache Hadoop

A biblioteca *Apache Hadoop* é um *framework* desenvolvido para ser escalável, desde um único servidor até milhares de máquinas em *clusters*, a biblioteca foi desenvolvida para detectar e lidar com falhas no nível da camada de aplicação, permitindo assim, um serviço com altas taxas de disponibilidade sem depender totalmente de um hardware confiável. A *Hadoop* utiliza um modelo de programação simples e uma linguagem de programação de alto nível chamada *Pig*, onde as execuções paralelas de programas complexos são quase triviais. Outro ponto relevante é a capacidade de otimização e extensão do sistema e da linguagem utilizada por parte dos programadores, dentro dos módulos pertencentes à solução do *Apache Hadoop* temos, por exemplo, o *Hadoop Map-Reduce* que utiliza o modelo de *Map-Reduce* para quebrar grandes conjuntos de dados em conjuntos menores, e então aplicar as operações de processamento, bem como o módulo *Hadoop Distributed File System* (HDFS™) responsável por gerenciar um sistema de arquivos

distribuídos, capaz de fornecer um acesso de alto desempenho para os dados de aplicativos, além de serem tolerantes as falhas de sistema como um todo.

O *framework Apache Hadoop* é largamente utilizado para processamento de *Big Data* em diversas empresas de grande porte como, Yahoo, Google e IBM, principalmente para obtenção e processamento de informações relevantes em grandes massas de dados.

Com a proposta de processamento utilizando o *Apache Hadoop* podemos utilizar características que envolvem processamento distribuído em *clusters* de forma eficiente para propor também uma solução no nível de armazenamento, como citado anteriormente, o próprio *Apache Hadoop* contém um módulo de gerenciamento de arquivos em *cluster* de forma distribuída, o HDFS.

Sendo assim, para obter o processamento de *Big Data* provenientes da *Internet Of Things* será necessário adotar uma arquitetura baseada em *clusters* que ofereça suporte para o *framework* do *Apache Hadoop*, uma arquitetura comumente utilizada para isto e com capacidade de elasticidade de recursos, é a utilização de *Cloud Computing* (Computação em Nuvem).

2. Armazenamento

Para o armazenamento é necessário encontrarmos uma solução capaz de ser facilmente integrável com o *Hadoop* e que seja funcional utilizando *Cloud Computing*.

Milhões de dados essenciais são perdidos diariamente devido às tecnologias tradicionais de armazenamento, como por exemplo, bancos de dados relacionais não distribuídos. Além da baixa capacidade de integração com serviços em *Cloud*, bancos de dados relacionais são de difícil análise quando lidamos com *Big Data*, inferências sobre os dados e análises semânticas exigem muitas vezes a necessidade de junções de diversas fontes de dados, fazendo com que sejam realizadas milhares de operações de junção de tabelas. Além de consultas cada vez mais complexas para recuperar os dados espalhados, outro ponto que deve ser levado em consideração é a baixa capacidade dos bancos de dados relacionais no armazenamento de dados heterogêneos. Esses dados são atrelados a uma modelagem do banco de

dados em si, sendo assim complexo e custoso realizar a alteração desta modelagem para a inclusão, por exemplo, de novos tipos de dados.

Dados os fatores citados, necessitamos de uma proposta de armazenamento capaz de lidar com os três pontos principais de *Big Data*, com capacidade de armazenamento escalável. Atualmente, *Big Data* considera volume de dados que podem chegar até Terabytes, em um futuro não muito distante *Big Data* irá considerar volumes de dados a partir de Petabytes. Além disto, a proposta deve ser capaz de dar suporte ao processamento desses dados, preferencialmente utilizando das vantagens do *Apache Hadoop*, com uma modelagem capaz de facilitar tanto as consultas quanto as inferências sobre os dados, ou seja, uma modelagem adaptável capaz de suportar dados heterogêneos de forma simples.

Uma solução capaz de suportar todos esses requisitos são os bancos de dados *NoSQL* que serão descritos em um tópico posterior.

B. *Cloud* Computing

Na ciência da computação, *Cloud Computing* (Computação em Nuvem) é sinônimo de computação distribuída através de uma rede, podendo esta ser a Internet ou não. Ou seja, indica a capacidade de executar programas ou aplicações em diversos computadores conectados ao mesmo tempo, muitas vezes estes computadores são conhecidos como *Clusters*.

A gestão e análise eficaz dos dados em larga escala, representa um interessante e crítico desafio. As tecnologias atuais baseadas, por exemplo, em computação em *Grid* e em *Cloud* [10], têm como objetivo prover grandes capacidades de processamento oferecendo uma visão única do sistema, fornecendo uma infraestrutura virtual, integrando dispositivos de monitoramento, armazenamento, ferramentas de análise, plataformas de visualização e acesso eficiente dos dados por parte do usuário. Ou seja, a virtualização das tecnologias a fim de maximizar a utilização dos recursos computacionais, permitindo que os usuários não precisem manter recursos caros tanto de hardware como de software.

Cloud Computing é uma poderosa arquitetura para realizar processamentos complexos e em larga escala, em uma era onde a quantidade de dados gerados ocorre de forma intensiva, é essencial que adotemos um processamento paralelo a fim de obtermos uma análise desses grandes volumes de dados, em um tempo aceitável. O uso de técnicas e algoritmos para processamento em paralelo são a chave para obter melhor escalabilidade e desempenho, atualmente os modelos paralelos mais populares incluem MPI (*Message Passing Interface*) [11], Unidades de processamento gráfico de propósito geral (GPGPU) e o já citado *Map-Reduce* assim como suas variações como o *MapReduce-like*.

Big Data e *Cloud Computing* são tecnologias que estão sendo desenvolvidas [12] e estão quase sempre relacionadas às dificuldades de processamento encontradas ao utilizar infraestruturas convencionais para o processamento de *Big Data*.

Podemos resumir as vantagens da computação em nuvem em três partes: Elementos Econômicos, Arquiteturais e Estratégicos.

1. Elementos econômicos possuem as vantagens do custo proporcional ao uso dos recursos de processamento, ou seja, o usuário tem a vantagem de pagar pelo que está utilizando (*Pay-as-you-go*), dessa maneira os recursos são desperdiçados o mínimo possível. Conforme o usuário necessite de maiores capacidades de processamento e armazenamento as soluções em *Cloud* por serem escaláveis, se adaptam rapidamente e mais recursos passam a ser alocados, aumentando assim o custo proporcionalmente ao que está sendo utilizado (*Pay-as-you-grow*).

2. Na visão dos elementos arquiteturais, as vantagens de utilizar *Cloud* estão na capacidade de abstração dos recursos físicos, componentes básicos como capacidade de armazenamento, CPUs e capacidade de rede utilizada fornecidas como um serviço de forma unitária. O usuário passa a enxergar apenas unidades que representam capacidades de processamento e armazenamento teoricamente infinitas [13]. Fisicamente os recursos de hardware são desconhecidos pelo usuário, criando uma espécie de grande máquina virtual com recursos escaláveis

sob demanda. Outro ponto relevante para a escolha de *Cloud,* para o uso na nossa proposta, ocorre devido ao fato da *Cloud* em si ser baseada em clusters. Este fato pode ser aproveitado tanto para aumentar a eficiência de processamento e armazenamento, mas principalmente para permitir a capacidade do sistema de lidar com um número possível de milhares de consultas concorrentes a partir de diversos dispositivos diferentes. Bem como com milhares de novos dados destes mesmos dispositivos ou de outros, de forma rápida e sem perda de informações.

3. E por fim, no campo dos elementos estratégicos, os desenvolvedores e usuários de serviços em *Cloud* podem focar os seus esforços no objetivo de negócio da aplicação, ou seja, oferecer muitos serviços virtuais, não é necessária dedicação a camadas de baixo nível sendo necessário na maioria das vezes maior dedicação a camada de aplicação, o que aumenta a produtividade e o foco no objetivo da aplicação.

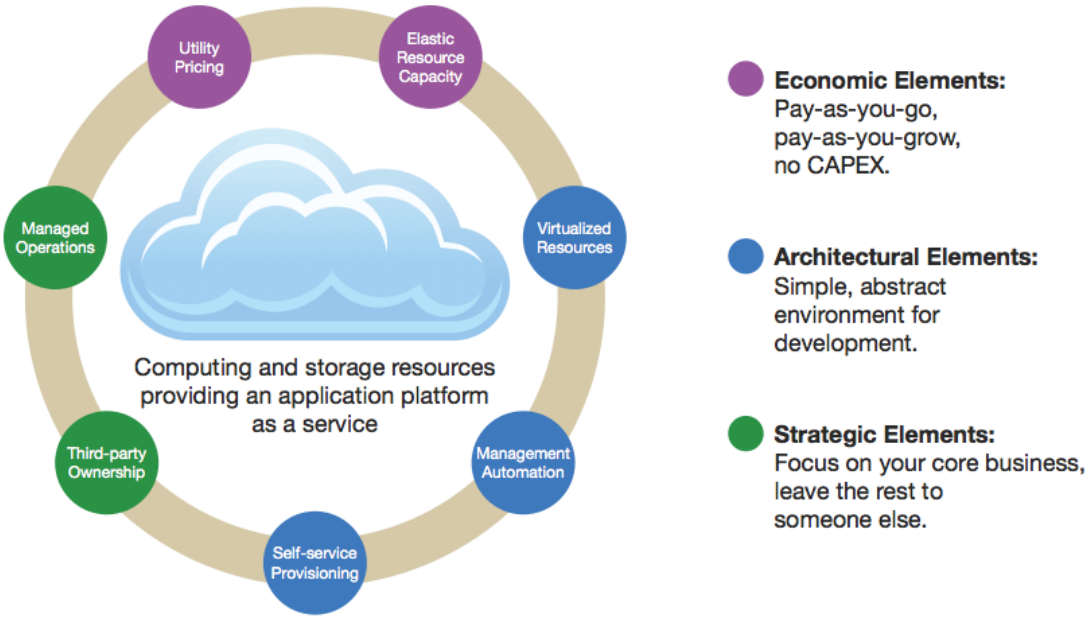

Figura 2 - Visão Geral Cloud Computing

Para atingir os objetivos nas análises de *Big Data*, muitas empresas que fornecem serviços em *Cloud Computing* vêm desenvolvendo diversas ferramentas, a *Amazon Web Services* (AWS) apesar de não ser um serviço gratuito é de fácil utilização e o seu sistema monetário é atrativo tanto para usuários iniciantes que necessitam de pouca capacidade de processamento quanto para usuários avançados com grandes necessidades de processamento. Aumentando o custo de forma escalável juntamente com os recursos utilizados, o *Eucalyptus* usa máquinas virtuais para realizar o controle e monitoramento dos recursos com API compatível com a AWS [14], *OpenNebula* [15] fornece integração com vários ambientes, fornecendo características ricas e interoperabilidade para produzir desde *Cloud* privadas até públicas.

IV. Banco de Dados

Big Data vem se tornando cada vez mais relevante. Aplicações de *Intelligence Business*, Análise de empresas, Gestão de relacionamento com o Cliente, processamento de documentos, redes sociais e *Cloud Computing* são exemplos de domínios que geram uma grande quantidade de dados e, por isto, exigem uma escalabilidade horizontal com suporte a armazenamento de informações relevantes, sejam estas estruturadas ou não.

Analisar os diversos tipos de bancos de dados existentes e procurar o que melhor se adapte as características tanto do ambiente de *Cloud Computing*, quanto às características de *Big Data* é uma tarefa importante no contexto deste trabalho de graduação.

A. Tipos de Banco de Dados

Existe uma diversidade de tipos de banco de dados, podemos citar alguns principais:

Flat File Database

Neste tipo de BD os dados são armazenados em um único arquivo de forma binária ou até em um texto simples, contendo uma coleção lógica de registros sem nenhuma relação estruturada entre eles. Bancos de dados *Flat File* se tornam inadequados quando necessitamos adicionar novos tipos de dados, incluir a segurança dos dados e expansão do banco. Além de ser necessária uma lógica adicional para que os dados registrados nestes grandes arquivos sejam interpretados corretamente.

1. Relacional Database Management System – RDBMS

RDBMS surgiu como uma solução para os desafios enfrentados por banco de dados *Flat File*. O armazenamento dos dados no RDBMS ocorre utilizando tabelas, representado os dados por colunas e linhas. Por exemplo, em uma linha podemos armazenar uma entidade como um cliente, e em cada

coluna desta entidade pode ser armazenado informações do mesmo, como telefone e endereço.

A maior vantagem deste tipo de banco de dados é a habilidade de descrever relacionamentos entre os dados utilizando junções entre tabelas distintas, chaves estrangeiras e etc, além de um armazenamento estruturado, utilizando índices nas colunas e linhas é possível recuperar as informações de forma muito eficiente, além de ser capaz de adaptar a expansão dos dados.

RDBMS, comumente utilizam uma linguagem de programação para consultas e atualizações dos dados, conhecida como *Structured Query Language* (SQL).

Bancos de dados comerciais do tipo RDBMSs, não são os mais adequados para processamento de dados com características de *Big Data*. A taxa com que os dados estão sendo gerados e a natureza não estruturada dos mesmos ultrapassou a capacidade dos bancos de dados relacionais. Por exemplo, se usuários ou dispositivos realizam uma atividade no banco de dados a cada 40 segundos, apenas isto já levaria alguns RDBMSs ao seu limite, pois seriam 5000 usuários/dispositivos simultaneamente por instância; Imaginem então, o que acontece quando estes números são multiplicados por 10 que seria equivalente a uma aplicação real gerando e processando *Big Data*;

As arquiteturas clássicas têm como possível gargalo a dificuldade em tornar os bancos de dados convencionais escaláveis e mais baratos, além de manter um esquema rígido na modelagem dos dados. Escalabilidade e flexibilidade na modelagem são requisitos essenciais no contexto de *Big Data*.

A taxa com que os dados estão sendo gerados através de um grande número de usuários ou dispositivos simultâneos exige, que para obtermos uma análise destes dados em tempo hábil é imprescindível o uso de processamento distribuído, envolvendo grande número de servidores e aplicações de manuseio de *Big Data*.

Assim, necessitamos mudar o foco para a adoção de um banco de dados diferente dos RDBMSs tradicionais, o que nos leva ao estudo de bancos de dados *NoSQL*.

2. NoSQL Database

Sistemas de banco de dados *NoSQL* [5] abordam principalmente os desafios de escalabilidade e desempenho, problemas não solucionados nos RDBMS tradicionais. Com capacidade de gerenciar um grande volume de dados não relacionais e não estruturados, com acessos rápido as informações e disponibilidade dos dados mesmo sob condições degeneradas do sistema, devido a vantagens de particionamento dos dados, fornecendo escalabilidade quase infinita e grande desempenho para os casos de uso intensivo de dados, levam os banco de dados *NoSQL* a ter o mesmo poder de consulta que os databases SQL com o diferencial de incluir mais recursos [16].

A escolha do banco *NoSQL* adequado para uma determinada aplicação não é trivial, alguns fatores importantes devem ser levados em consideração nesta escolha:

1. Escalabilidade;
2. Desempenho;
3. Disponibilidade;
4. Facilidade de uso.

Exemplos de banco de dados *NoSQL* incluem o *MongoDB* que é utilizado em aplicações do *Foursquare, Sourceforge, The New York Times, Disney* e outros. Com foco no processamento em larga escala utilizando *frameworks* como o já citado *Apache Hadoop* com o *Map-Reduce* temos o *Cassandra*, largamente utilizado no *Facebook, Twitter, Cisco, Digg, Reddit* e etc.

A seguir serão descritas as categorias dos bancos de dados NoSQL.

a) Categorias de NoSQL Database

Dentre os banco de dados *NoSQL*, existem diversas categorias com características e abordagens distintas para o armazenamento, uma classificação básica destas categorias estão relacionadas principalmente com o modelo de dados utilizado, as principais categorias são:

(1) Chave/Valor

Os dados são armazenados sem esquema pré-definido, no formato de pares Chave/Valor, onde temos uma Chave que é responsável por identificar o dado e seu valor que corresponde ao armazenamento do dado em sí. Sem dúvida este é a categoria de *NoSQL* mais simples para o processamento em ambientes de *Big Data*, oferecendo grande flexibilidade e escalabilidade.

Entretanto, Bancos de dados Chave/Valor não oferecem todas as características do modelo ACID, sendo necessário que os desenvolvedores levem em consideração a localização, replicação e tolerância a falha neste banco de dados, além de que banco de dados Chave/Valor não contém tipos, o que resulta na armazenagem dos dados quase sempre no formato de Strings. Deste modo, dependendo da quantidade de usuários/dispositivos no sistema, manter um controle dos dados na forma de Chave/Valor se torna desafiador.

Propostas vêm surgindo visando corrigir os problemas desta categoria, por exemplo, um banco de dados de código aberto amplamente utilizado é o *Riak* [17], desenvolvido pela Bash Technologies, com suporte a processamento paralelo utilizando o *Map-Reduce*.

(2) Orientado a Colunas

Este tipo de categoria se tornou conhecida devido a implementação da Google (*Google's BigTable*), são muito similares em um nível abstrato aos RDBMSs, com algumas diferenças mais do ponto de vista conceitual, os dados são armazenados em famílias de colunas, com a adição de alguns atributos dinâmicos. Por exemplo, são utilizadas chaves estrangeiras mas que apontam para diversas tabelas diferentes, ou seja este banco de

dados não é relacional e por este motivo eles não tem um gerenciamento que identifiquem as tabelas como no RDBMS.

Bando de dados orientados a colunas foram criados para processar um grande número de dados de forma distribuída através de diversas máquinas, entretanto é mais difícil de serem compreendidos, apesar de parecerem superficialmente com RDBMS, a lógica envolvida é mais complexa, o que pode ser um empecilho para o desenvolvimento de aplicações.

Os bancos desta categoria mais utilizados são: o *Cassandra, Hadoop/HBase* e o *Google's BigTable*.

(3) Orientado a Documentos

O conceito principal nos bancos de dados orientados a documentos, os dados são tratados como objetos independentes no formato de documentos. Cada documento contém uma informação única pertinente a um único objeto, mantendo a estrutura dos objetos armazenados. Em geral, todos os dados são assumidos como documentos, que por sua vez são encapsulados e codificados em algum formato padrão, podendo ser estes formatos *XML, YAML, JSON, BSON,* assim como formatos binários como PDF e formatos do *Microsoft Office*.

A popularidade dessa categoria ocorre principalmente devido à elegância na programação tanto pelas vantagens de escalabilidade como de desempenho. Entretanto, se esta categoria consegue ser superior em relação aos tradicionais SQL, quando considerado banco de dados em larga escala, ainda é uma questão em aberto.

Exemplos deste tipo de banco de dados, podemos citar o MongoDB [18], Apache Couch DB [19] e ClusterPoint [20].

(4) Grafos

Banco de dados baseado em grafos, armazenam os dados em uma estrutura de nós, arestas e propriedades, com os nós representando as

entidades em si, as arestas representando os relacionamentos entre os nós e as propriedades representando os atributos das entidades, podendo estas serem representadas tanto nos nós quanto nas arestas do grafo. Essa categoria é similar a RDF (*Resource Description Protocol*) [21] em relação ao armazenamento dos dados no formato de relacionamentos e entidades, assim como RDF podem ser considerado banco de dados em grafo.

Portanto, em vez de utilizar uma estrutura rígida como no SQL, um modelo em grafo pode ser muito flexível, pois a modelagem dos dados passa a ser adaptável e crescente conforme os dados vão sendo armazenados, sem modelagem pré-definida e com propriedades dinâmicas, permitindo assim a distribuição do grafo através de diversas máquinas. Contribuindo para a escalabilidade, tolerância a falhas e consequentemente a capacidade de processamento que esperamos para análise de *Big Data*.

Esse tipo de banco de dados utiliza teorias matemáticas dos grafos, comumente utilizadas nas mais diversas aplicações, com uma modelagem mais natural dos dados. Sendo possível realizar consultas com um alto nível de abstração considerando as vantagens da própria estrutura do grafo representar a modelagem dos dados. Por esses motivos, esta categoria é indicada para aplicações em redes sociais e que necessitam de processamento inteligente como inferências a partir dos dados armazenados. Possui capacidade para altas cargas de trabalho com diversos usuários/dispositivos utilizando o banco de dados ao mesmo tempo. Podemos citar por exemplos aplicações na área de transportes [22], *Bussiness Network* [23], aplicações em bioinformática, quimio-informática e interações entre proteínas [24]. Outro exemplo importante é o *The Web of Data*, com 31 bilhões de triplas e 466 milhões de links [25] demonstra a capacidade de armazenar grandes conjuntos de dados.

Bancos de dados baseados em grafos surgiram há poucos anos, apesar da tecnologia ser relativamente nova, já existem alguns estudos na área.

Tabela 1 - Banco de dados - Tamanhos (Vicknait, C. et al. 2010)

Database	#Nodes	Data Type	MySQL Size	Neo4j Size
1000int	1000	Int	0.232M	0.428M
5000int	5000	Int	0.828M	1.7M
10000int	10000	Int	1.6M	3.2M
100000int	100000	Int	15M	31M
1000char8k	1000	8K Char	18M	33M
5000char8k	5000	8K Char	87M	146M
10000char8k	10000	8K Char	173M	292M
100000char8k	100000	8K Char	1700M	2900M
1000char32k	1000	32K Char	70M	85M
5000char32k	5000	32K Char	504M	406M
10000char32k	10000	32K Char	778M	810M
100000char32k	100000	32K Char	6200M	7900M

Em [26] encontra-se uma análise comparativa entre RDBMS e banco de dados em grafo, na tabela 1 pode-se observar que apesar dos bancos de dados em grafo serem de 1.5 a 2 vezes maiores que os RDBMSs, levam vantagens quando analisados o tempo de consulta, com performance de 10 vezes maior que os RDBMSs convencionais como demonstrado na tabela 2.

Tabela 2 - Resultado do tempo de consulta em milissegundos (Vicknait, C. et al. 2010)

Database	MySQL S4	Neo4j S4	MySQL S128	Neo4j S128	MySQL S0	Neo4j S0
1000int	38.9	2.8	80.4	15.5	1.5	9.6
5000int	14.3	1.4	97.3	30.5	7.4	10.6
10000int	10.5	0.5	75.5	12.5	14.8	23.5
100000int	6.8	2.4	69.8	18.0	187.1	161.8
1000char8K	1.1	0.1	21.4	1.3	1.1	1.1
5000char8K	1.0	0.1	34.8	1.9	7.6	7.5
10000char8K	1.1	0.6	37.4	4.3	14.9	14.6
100000char8K	1.1	6.5	40.9	13.5	187.1	146.8
1000char32K	1.0	0.1	12.5	0.5	1.3	1.0
5000char32K	2.1	0.5	29.0	1.6	7.6	7.5
10000char32K	1.1	0.8	38.1	2.5	15.1	15.5
100000char32K	6.8	4.4	39.8	8.1	183.4	170.0

Em [27] encontra-se uma análise quantitativa que compara os modelos e características fornecidas por nove bancos de dados diferentes. Em [28] e [29], alguns bancos de dados são analisados a partir de uma visão de desempenho, considerando operações de criação de grafos e operações de consulta transversais (as que necessitam percorrer todo ou uma parte do grafo a partir

de um determinado nó)., Finalmente, o trabalho recente da *Tinkerpop* [30] com a criação de um *framework* capaz de facilitar o uso desse tipo de bancos de dados.

O *Neo4J* [31] atualmente é o mais utilizado por empresas como *Adobe, HP* e *Huawei*, além de outros banco de dados como o *Titan* [32]*, AllegroDB* [33] *e o Dex* [34]. Cada um destes contém suas funcionalidades e características específicas, por exemplo, o *Titan* foi desenvolvido para funcionar de forma distribuída através de várias máquinas, outros provêm nativamente sua própria linguagem de consulta, como o *Cypher* no *Neo4J* e o *Sones 8* com o *GraphQL*, sendo estas linguagens um dos grandes diferenciais em relação ao tradicional SQL. Alguns exemplos de linguagens para *NoSQL* são:

- UnQL(Linguagem de consulta não estruturada);
- SPARQL (SPARQL Protocol e linguagem de consulta baseada em *Resource Description Framework - RDF*);
- GQL (*Google Query Language*);
- SONES (*Graph Query Language*);
- GREMLIN (Linguagem de consulta transversal).

Em [35], foi realizado um estudo comparativo entre diversos banco de dados atuais baseados em grafos, neste trabalho foi criada uma ferramenta em Java open-source chamada *Graph Database Benchmark* (GDB) que seria responsável por realizar um *benchmarking* de forma distribuída em algumas soluções de grafos já citados. Neste artigo para analisar o impacto do tamanho do grafo no desempenho, foram realizados os testes utilizando diversas cargas de trabalho em grafos de tamanhos variados Primeiro foi feita uma análise com 250.000 vértices gerando aproximadamente 1.250.000 arestas e posteriormente considerou-se grafos com 500.000 vértices gerando aproximadamente 2.500.000 arestas.

Foram analisados o desempenho em quatro banco de dados, todos mantendo as suas configurações padrões:

1. Neo4j - Versão 1.9M05
2. Titan - Versão 0.3, utilizando conjuntamente com o *Cassandra* em *Embedded Mode*, o que permitiu um maior desempenho, pois tanto o

Titan quanto o Cassandra estariam rodando na mesma *Java Virtual Machine* (JVM)

3. OrientDB – Versão 1.3
4. DEX – Versão 4.7

Nas análises pode-se observar que alguns bancos de dados se sobressaíram em algumas características:

1. O *DEX* e o *Titan-Cassandra* se comportaram de forma similar no teste de adição de vértices, escalando particularmente bem quando os números de clientes aumentaram e também se destacando quando realizado testes de consulta de vértices e atualização intensiva;
2. Considerando apenas como carga de trabalho a leitura, o *Neo4J, Titan* com *BerkeleyDB* e *OrientDB*, sofreram uma drástica degradação de performance, enquanto novamente o *DEX* e o *Titan-Cassandra* obtiveram resultados muito mais interessantes do que os outros banco de dados.

Em [27] foram realizados alguns testes comparativos no ambiente de *Cloud*, e foram analisados as performances de cinco banco de dados baseados em grafo: o *AllegroGraph, Fuseki, Neo4J, OriendDB e Titan* utilizando também o *Apache Cassandra* como *backend*. Para realizar esta análise de desempenho foi utilizado a proposta de *benchmark* do artigo, o *XGDBench* utilizando o ambiente de *Cloud*, *Tsubame* versão 2.0 HPC [36]. Nas análises de desempenho realizadas neste experimento o *Titan* foi analisado utilizando seis nós, onde em um nó foi executado o servidor *Rexster* versão 2.1.0 [37] que faz parte do *framework Tinkerpop*, o *Apache Cassandra* sendo executado em quatro nós e o nó subsequente executando a ferramenta de *benchmark* o *XGDBench*. Alguns pontos neste artigo foram limitantes nas análises, entre eles a utilização de uma interface de comunicação entre a aplicação e o banco de dados do tipo HTTP/REST, o que contribuiu como um gargalo para a escalabilidade de alguns bancos de dados, contudo, a capacidade nos aspectos de escalabilidade do *Titan* foi confirmada.

A partir das informações obtidas e notando a capacidade de alguns banco de dados operarem em ambientes de *Cloud* como os, *NuvolaBase* [38] , *Cloud Graph* [39], *Heroku* [40] e outros, percebemos que seria um bom indicativo escolher um banco de dados baseado em grafo para a utilização na nossa proposta. Filtramos as nossas escolhas até chegar entre dois bancos de dados, o *DEX* e o *Titan-Cassandra*. Após mais algumas análises que serão discutidas nos tópicos consequentes, escolhemos para a nossa proposta o *Titan* juntamente com o *Cassandra.*

Na tabela 3, podemos verificar um comparativo entre as categorias de banco NoSQL, resumindo os tópicos discutidos anteriormente.

Tabela 3 - Comparativo entre as categorias de banco NoSQL [6]

	Key Value Stores	**Column Family Databases**	**Document Databases**	**Graph databases**
Based on	Dynamic Hash Tables, Dynamo DB	Google's Bigtable	Lotus Notes, encoding include JSON, XML	Euler's Graph Theory
Data Model	Key/Value pairs	Columns	Key/Value Collections	Graph structure-Nodes, Edges and Properties
Applicability	Handling massive load	Distributed file systems	Web applications, full text searches and updates, information ranking	Semantic web, Social Networks, Intelligent Agencies
Advantages	Simple and easy to implement	Fast querying of data, storage of very large quantities of data	Accepts partially complete data, allows efficient querying	Easy scaling of complex data across distributed systems.
Disadvantages	Inefficient in querying/ updating part of a database	Very low-level API	No standard query language	Traversal of entire graph to give correct results
Examples	Redis,Project Voldermort	Cassandra, HBase	MongoDB, CouchDB	Neo4J, InfoGrid
Data Model	Key/Value pairs	Columns	Key/Value Collections	Graph structure-Nodes, Edges and Properties

Na tabela 4, podemos observar um comparativo geral entre os principais tipos de banco de dados aqui citados.

Tabela 4- Sumário Flat File, RBDMS e NoSQL [6]

	Flat File Database	RDBMS	NoSQL
Data Model	Flat File	Tables	Columns, Graph, Document, Key/Value
Schema	Schema-less	Fixed Schema	Schema-less
Query Languages	CQL	SQL	API calls, JavaScript and REST
Integrity Model	None	ACID	CAP, BASE
Applicability	Any	Relational and transactional data	Non-relational data
Security	No security	Limited security mechanisms, vulnerable to SQL injection	Authorisation and authentication weaknesses, no encryption, Multiple interfaces increase attack surface.
Advantages	Simpler to use, Less expensive, suited for small scale use	Ensures data integrity between transactions, better security, supports medium to larger sized organisations, provides backup and recovery controls	Can cater for Big Data, unstructured data and distributed systems
Disadvantages	No support for multi-user access, redundancy and integrity problems	Expensive and difficult to manage in distributed systems, Complex and difficult to learn, not suitable for unstructured data	Security is a concern (no encryption), lack of standard query language, Too many varied databases thus no single solution for different purposes
Examples	MsDOS	Oracle, Postgres, MySQL, Microsoft SQL Server	MongoDB, Cassandra, Neo4J

V. Titan

O *Titan* é um banco de dados baseado em grafos distribuído criado pela *Aurelius* com o objetivo de processar e representar grafos com mais de 100 bilhões de arestas, utilizando técnicas inovadoras de compressão de arestas, *data layout* e índices *vertex-centric*. O *Titan* é capaz processar e representar estruturas de grafos através de diversas máquinas em *clusters*, escalando milhares de transações concorrentes, com capacidade de lidar com travessias complexas de grafos em tempo real (*Online Transaction Processing* ou Processamento de Transações em Tempo Real – OLTP), e permitindo a análise de um grande volume de dados sob múltiplas perspectivas (*On-line Analytical Processing* - OLAP) em blocos de processamento (*Batch*), além de ser gratuito e de código aberto.

Titan é capaz de lidar com o *tradeoff* entre escalabilidade e complexidade/interligação dos dados, principalmente para aplicações na área de *Big Data*, que necessitam tanto de suporte para grande volume de dados quanto liberdade na modelagem e descrição dos dados. A característica principal do Titan é a otimização no armazenamento e consulta em grafos. Os grafos podem conter bilhões de vértices e arestas distribuídos através de várias máquinas. O Titan mantém replicação dos dados para obter requisitos de desempenho e tolerância a falhas, oferecendo suporte para centenas de usuários que podem realizar operações e consultas complexas de forma concorrente.

Em sua versão mais recente, o *Titan* oferece suporte a grafos de tamanho teoricamente "infinito" em conjunto com um número "ilimitado" de usuários, através de sua mecânica de computação de grafos de forma distribuída. Portanto, o *Titan* consegue oferecer suporte para OLAP através do *Apache Hadoop* e é distribuído gratuitamente via licença *Apache2*, implementando nativamente o *framework Tinkerpop. Rexster* Além de permitir busca geográfica, por conjunto numérico e texto, através de *ElasticSearch* [41] e do *Apache Lucene* [42].

O *Titan* pode ser configurado em diversos ambientes, entre eles podemos citar o de grafos baseado em memória, onde o grafo é carregado na memória RAM do sistema, estando atrelado ao tamanho máximo da mesma, com vantagens de alta velocidade de processamento devido às características de leitura e escrita neste tipo de memórias serem mais velozes que a memória em disco. O Titan também possui algoritmos e visualizações de grafos especializados, exemplos de banco de dados atuantes neste tipo de ambiente são: o *iGraph*, *Jung* e *Networkx*.

Outra opção é a utilização do banco de dados em disco rígidos, apesar de a velocidade ser consideravelmente menor do que os bancos de dados em memória têm a vantagem do tamanho do grafo estar limitado ao tamanho do disco local, o que por ser mais barato, pode ser várias vezes maiores que um grafo em memória. Exemplos de banco de dados neste ambiente são: o *Neo4j*, *DEX*, *OrientDB* e *InfiniteGraph*.

E por fim, os bancos de dados em *clusters*, neste ambiente os grafos são distribuídos nas memórias de diversos *clusters*, e consequentemente limitados a memória de cada *cluster*. Entretanto como os grafos são particionados o tamanho dos mesmos podem se tornar “infinitos”, dependendo apenas da quantidade de *clusters* que estão contidos. Além destas, outras vantagens são os algoritmos otimizados para memória, a não limitação de tamanho como nos ambientes de memória com apenas uma máquina. Portanto, este ambiente é o mais promissor para a atuação do *Titan*, e também de outros bancos de dados como: o *Giraph*, *Hama* e *GoldenOrb*.

Enquanto os bancos de dados do tipo RDBMS utilizam transações baseadas apenas nos princípios ACID (Atomicidade, Consistência, Isolamento e Durabilidade), os bancos de dados *NoSQL* fazem uso de algumas caracteríscitcas distintas, como o teorema CAP (Consistency, Avaliability and Parition Tolerance) ou BASE (*Basically Available*, *Soft state*, *Eventual consistency*). No caso do *Titan*, o mesmo oferece suporte para características ACID e Consistência eventual.

Serviços que oferecem consistência eventual são classificados como BASE em contraste com as garantias tradicionais do ACID.

O modelo ACID funciona muito bem nos tradicionais RDBMSs, entretanto não funciona muito bem quando disponibilidade e desempenho são extremamente importantes, como em sistemas distribuídos. Um modelo mais conhecido é o BASE, juntamente com banco de dados *NoSQL*, surgiu como proposta para resolver os desafios enfrentados pelo modelo ACID, principalmente, quando aplicado a algumas características de *Big Data* como, por exemplo, a natureza não estruturada dos dados e a grande quantidade de usuários que estarão realizando transações no banco de dados de forma concorrente. Portanto, em alguns casos BASE pode ser uma alternativa mais eficiente que ACID.

Descrevendo melhor o acrônimo BASE, temos:

- Disponibilidade Básica (*Basically Available*):
    - Garante que o banco de dados esteja essencialmente acessível mesmo que parte do sistema falhe, utilizando técnicas conhecidas

como “*sharding*” ou simplesmente particionando os dados através de diversas máquinas, com replicação dos mesmos. Resultando em uma alta disponibilidade, independente das falhas possíveis que venham a ocorrer.

- Soft State (*Soft state*):
    - Permite que transações ocorram mesmo que as atualizações levem tempo para se propagarem através de todos os dados devido a erros ou distúrbios no sistema. Com o controle de consistência passado para a camada de aplicação (diferente do ACID), algumas inconsistências podem ser geradas e toleradas até certo nível, surgindo então outra característica a Consistência Eventual.
- Consistência Eventual (*Eventual consistency*):
    - Os sistemas das características BASE têm seus requisitos de consistência relaxados no final de cada operação, garantindo apenas que os dados serão armazenados em um estado porterior de uma forma consistente.

Dependendo das necessidades da aplicação e do banco de dados que será utilizado em seu *backend*, o *Titan* consegue se adaptar a diversos níveis de consistência, disponibilidade, escalabilidade, replicação e persistência..

O *Titan* oferece diversos bancos de dados para trabalhar em conjunto como seu *backend*, são eles: *Apache Cassandra*, *Apache HBase*, *Oracle BerkeleyDB* e *Akiban Persist*, permitindo assim que o desenvolvedor tenha uma maior customização do sistema, podendo modificar as características do mesmo apenas modificando a escolha do banco de dados em *backend*, criando assim uma aplicação portável para diversos ambientes.

Segundo o teorema de CAP, sistemas distribuídos como o *Titan* não podem oferecer as três características de Consistência, Escalabilidade e Disponibilidade ao mesmo tempo, estando limitados assim a apenas duas destas [43].

A tabela a seguir demonstra a capacidade de configuração do *Titan* na escolha do *backend*, de acordo com as necessidades do usuário.

Tabela 5 - Comparação entre Backends

Storage Backend Comparison

Name	`storage.option`	Consistency	Availability	Scalability	Replication	Persistence
Cassandra	*cassandra*	eventually consistent	highly available	linear scalability	yes	disk
HBase	*hbase*	vertex consistent	failover recovery	linear scalability	yes	disk
BerkeleyDB	*berkeleyje*	ACID	single point of failure	single machine	HA mode available	disk
Persistit	*persisit*	ACID	single point of failure	single machine	none	disk

Como visto no tópico anterior, o *Titan* demonstrou capacidade equivalente e em alguns pontos até mais eficiente do que seus concorrentes, como o *Neo4j* e o *DEX*.

Foi realizada uma avaliação do Titan pela *Aurelius* [44], onde o objetivo foi simular um teste de capacidade de processamento de transações por segundo. Foi utilizado um banco de dados do *Twitter* de 2009 e então carregado no *Titan* utilizando o *BatchGraph*, este grafo continha um total de 1.47 bilhões de arestas e 41.7 milhões de usuários, a partir deste banco de dados foi realizado quatro tipos de transações:

1. Criação de Contas (1%)
2. Publicação de novos "*Tweet*"s (15%)
3. Leituras de Fluxos (76%)
4. Recomendações (8%)
5. Seguir recomendações de usuários (30%)

Na avaliação deste teste, o *Titan* foi capaz de realizar 5.900 transações por segundo durante uma carga de trabalho normal, e 10.200 transações por segundo sob uma carga de trabalho de pico. O teste foi realizado utilizando os serviços de *Cloud* da *Amazon* EC2, contendo instâncias do Titan com o *Cassandra* como *backend* distribuídos entre os clusters, com o custo de apenas 11 dólares por hora (ou $96.360 por ano).

O que mostra novamente, que apesar da tecnologia ter sido disponibilizada em 2012, utilizando tecnologias já consolidadas e emergentes como o *Cassandra* e

Amazon EC2, o *Titan* foi capaz de realizar bem suas propostas, e provavelmente tornando-se uma das tecnologias que mais crescem no ramo.

VI. Apache Cassandra

O *Cassandra* [45] foi desenvolvido inicialmente pelo *Facebook*, sendo liberado como projeto *open-source* em 2008, baseado em banco de dados não relacionais, pertencente à família dos bancos de dados orientados a colunas, utiliza um sistema distribuído baseado nas tecnologias de armazenamento *DynamoDB* [46] unido com modelos do *Google's BigTable* [47], e desenvolvido com o objetivo de armazenar dados característicos de *Big Data*.

Atualmente utilizado por empresas como o *Twitter*, com um modelo *peer-to-peer*, *Cassandra* é um sistema tolerante a falhas devido à redundância dos dados em diversas máquinas, com escalabilidade horizontal superior a maioria dos bancos de dados existentes, pode permitir um menor custo e maior desempenho se comparado com banco de dados que escalam verticalmente. Cassandra pode garantir uma eficiência nos procedimentos de escrita dos dados, removendo assim os gargalos inerentes a este tipo de operação nos grafos que serão utilizados pelo *Titan*. Um *cluster* de máquinas contendo o *Cassandra* pode ser expandido sob demanda, simplesmente adicionando novas máquinas ao sistema, o que o torna uma excelente opção para os ambientes de *Cloud Computing*.

Outra característica importante no *Cassandra* é a capacidade de integração com o *Apache Hadoop* e o paradigma *Map-Reduce* para processamento em tempo real. Outras funcionalidades do *Cassandra*, incluem, o sistema distribuído próprio do *Hadoop*, o HDFS, que permite a replicação dos dados de forma configurável, onde cada réplica é alocada em uma máquina distinta no *cluster*, podendo gerar um cluster *Hadoop-Cassandra*.

Em [48], é apresentado uma avaliação do *Cassandra* quando utilizado em conjunto com a tecnologia *Hadoop Map-Reduce* onde são analisados vários aspectos do *Cassandra*, como por exemplo, localidade e distribuição dos dados, assim como escalabilidade, balanceamento de carga e desempenho em operações de entrada e saída de dados. As configurações que causaram maiores impactos no desempenho *Cassandra + Apache Hadoop* são

apresentadas, através de testes realizados no *Yahoo Cloud Serving Benchmark* (YCSB), um *benchmark* em ambiente de *Cloud*.

Por estes motivos, escolhemos o *Cassandra* em conjunto com o *Titan*, a fim de obtermos as características de Consistência, Disponibilidade, Escalabilidade, Replicação e Persistência dos dados, juntamente com Consistência Eventual (BASE), de forma que melhor atendesse as necessidades da nossa aplicação.

VII. TinkerPop Stack

Em 2010, *Tinkerpop* [30] iniciou um projeto *open source* sob licença BSD, o *Blueprints*, uma API em *Java* voltada para facilitar o uso com banco de dados baseado em grafos. *Blueprints* define um modelo particular de grafos, basicamente grafos direcionados, com arestas e vértices contendo propriedades. *Blueprints* também fornece uma interface básica com uma série de métodos que permitem a interação com os grafos, de qualquer tamanho, utilizados em diversos bancos de dados, como por exemplo, o *Titan, Neo4j, DEX, OrientDB* entre outros. *Blueprints* oferece métodos abstratos que permitem, por exemplo, remover, consultar e atualizar tanto vértices quanto arestas, a partir dos valores de atributos, sendo capaz de lidar com grafos contendo bilhões de arestas distribuídas em um *cluster*.

Alguns módulos importantes do *Blueprints* podem ser observados na Figura 3, sendo: o *Rexster*, que é um servidor REST (Representational State Transfer), voltado para o acesso remoto dos grafos, o *Gremlin* e o *Pipes* (utilizado para processamento dos dados em formato de fluxos). Além dos módulos *Furnace* e *Frames*, que serão descontinuados nas próximas versões do *Blueprints.*

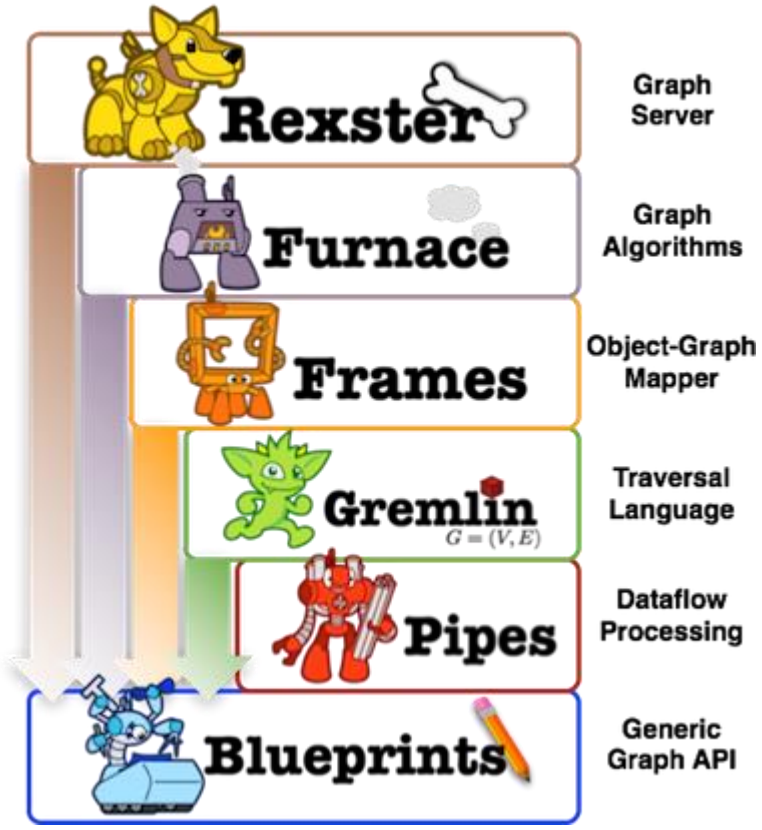

Figura 3 - Blueprints Stack [24]

O *Blueprints* permite gerar código portável, pois algumas configurações são realizadas de forma automática, tornando a troca do banco uma operação rápida e fácil, sem necessidade de adaptação nas camadas acima que utilizam o *Blueprints*. Tecnologia esta, que vem sendo de grande interesse em diversas propostas que utilizam banco de dados baseado em grafos.

A. *Rexster*

Rexster [49] é um servidor de grafos, que permite expor qualquer módulo do *Blueprints* através de uma API REST e de um protocolo binário denominado *Rexpro*, com esta funcionalidade qualquer linguagem de programação pode ser utilizada.

Nesse módulo também é oferecido uma interface web *The Dog House*, onde é possível realizar operações no banco de dados em um nível mais abstrato, com exibição tanto visual quanto estruturada dos dados. Esta interface fornece também um console para cada grafo individualmente, onde o usuário pode executar diversos comandos descritos em *Gremlin* de modo remoto.

Rexster também permite a expansão das suas funcionalidades através da criação de extensões, conhecida como *Rexster Kibbles*. Podemos citar o *SPARQL Kibble*, que permite a execução de consultas descritas em *SPARQL,* e o *Batch Kibble*, responsável por fornecer a possibilidade de executar diversas operações através da API REST, em grandes blocos de processamento (*Batchs*).

B. Pipes

Pipes [50] é um *framework* baseado em fluxos de processamentos em grafos. Um *Pipe* recebe como entrada vários dados em fluxo onde os mesmos podem ser separados, agrupados, transformados, etc., gerando um novo fluxo de dados resultante da operação realizada no *Pipe*.

Por ser composto por apenas um simples passo computacional, os *Pipes* podem ser agrupados gerando um processamento composto das informações, por exemplo, podemos ter um *Pipe* para selecionar todos os vértices que estão saindo de um determinado nó, e aplicarmos na sua saída o mesmo *Pipe* ou outro *Pipe*.

Existem diversas classes de *Pipe* e o próprio usuário pode definir o seu, fazendo com que este tipo de processamento de fluxo seja intuitivo e eficiente, como o comando a seguir:

```
g.v(1).out('knows').filter{it.age < 30}.name
```

onde ‘g’ representa o grafo e v os vértices. Neste exemplo, apenas o primeiro vértice do grafo nos interessa, os seguintes são os *Pipes*:

- *Out* – Retorna as arestas que saem do nó, contendo a propriedade ‘*knows*’;
- *Filter* – Filtra os resultados para retornar apenas aqueles que contêm a propriedade *age* menor que 30;
- *Name* – Retorna o valor contido no atributo “*name*”, apenas se o atributo existir.

C. Gremlin

Enquanto os RDBMS utilizam linguagens como SQL e JDBC (Java Database Connectivity), temos no *Blueprints* a utilização do *Gremlin.*

Figura 4 - Esquemático Gremlin [24]

Gremlin é uma linguagem Turing Completa[1] diferente do SQL, de domínio específico que realiza operações transversais complexas em banco de dados baseados em grafos multi-relacionais, chamados grafos de propriedades. *Gremlin* tem como base a linguagem *Groovy* que permite a utilização em conjunto com *Java*.

Gremlin permite realizar operações de adição ou remoção de vértices de arestas, manipulação de índices do grafo, pesquisas por elementos específicos, carregamento de uma base de dados em grafo a partir de URL ou de arquivo, e muitas outras operações. Utiliza uma sintaxe compacta, o que pode tonar a leitura dos comandos uma tarefa não trivial, como no caso de consultas que envolvem muitas propriedades e relacionamentos, veja o exemplo de comando a seguir.

```
g.v(1).out('knows').as('here').name.filter{it[0]=='v'}.back('her
e')
```

[1] Tem o poder computacional pleno, podendo realizar qualquer tipo de cálculo, desde que tenha memória para isso, ou seja, tem o mesmo poder de processamento de uma máquina de Turing.

Para o desenvolvedor, não apenas o desempenho da linguagem é importante, mas a capacidade de leitura e manutenção da mesma deve ser levada em consideração.

Em [51] foram realizadas diversas análises comparativas entre as linguagens de consulta voltadas para banco de dados baseados em grafos, comparando entre *Cypher, Gremlin* e o acesso nativo no *Neo4j*. Em um dos tópicos, as características do *Gremlin* são demonstradas e em alguns casos são mais eficiente que o *Cypher*, mesmo quando a infraestrutura não foi otimizada para o uso do *Gremlin*. Outro ponto comentado foi a vantagem do *Gremlin* realizar consultas transversais devido a facilidade para a escrita dos comandos, muitas vezes se assemelhando a um acesso nativo dos objetos.

VIII. Faunus

Titan pode suportar milhares de usuários executando operações complexas de forma concorrente em apenas um grafo massivo.

Faunus [52] é uma ferramenta de análises de grafos, que consegue gerar um grafo em larga escala, formado por relacionamentos simples, permitindo maiores vantagens quando é necessário operações globais no grafo, por exemplo, quando o usuário necessita realizar uma consulta transversal ou alguma operação mais complexa.

A partir do grafo massivo do *Titan*, o *Faunus* gera vários sub-grafos reduzidos com uma semântica rica e com relacionamentos simples. Os sub grafos são distribuídos em diversas máquinas em *cluster*, implementando álgebra multi-relacional como uma coleção de operações do tipo *Map-Reduce*. O *Faunus* consegue unir as vantagens do banco de dados baseado em grafo com as características de processamento distribuído eficiente utilizando o *Map-Reduce*. Estas características tanto para processamento de forma rápida utilizando o ambiente de *Cloud* ou para processamento de dados *off-line* em *batchs são úteis*.

Faunus oferece suporte ao *Map-Reduce* através do *Apache Hadoop*, incluindo também as vantagens do tipo de arquivo HDFS. Diversos tipos de arquivos de entrada e saída são suportados pelo *Titan*, dentre eles o *Tinkerpop Rexster, RDF Text Files, Hadoop Binary Files* além de *scripts* próprios.

O *Fulgora* é um módulo voltado para processamento de grafos em memória, com o objetivo de executar principalmente algoritmos de aprendizado de máquina e buscas iterativas nos grafos, utilizando fortemente o conceito de *threads* e as vantagens de velocidade que os bancos de dados em memória oferecem. Entretanto este módulo ainda encontra-se em fase inicial de desenvolvimento.

A solução da *Aurelius* (composta pelos módulos descritos anteriormente) para aplicações em banco de dados baseados em grafos são completas e integradas entre sí, fornecendo uma larga cobertura para os mais diversos problemas computacionais encontrados em *Big Data*.

IX. Esquema da Arquitetura

Após um estudo das tecnologias nas áreas de *Big Data* e *Cloud Computing*, e análises entre as vantagens e desvantagens de tecnologias já conhecidas e tecnologias recentes, foi definida a arquitetura proposta por este trabalho de graduação.

Basicamente, será utilizado o *Titan* como um banco de dados em *Cloud* (nuvem), devido as suas características de escalabilidade, suporte a múltiplos usuários de forma concorrente e principalmente por sua arquitetura em grafo que facilitará inferências e processamentos dos dados em *Big Data* provenientes da *Internet Of Things*.

Em conjunto com o *Titan* será utilizada a proposta da *Stack Blueprints*, e consequentemente os módulos inerentes à mesma, como o servidor *Rexster*, que disponibiliza uma interface de acesso a qualquer grafo pertencente ao *Titan.* Com o *Gremlin* será possível realizar diversas operações, com qualquer linguagem que o usuário desejar, através do *RexPro*, ou acessando o console Gremlin contido no *"The Dog House"*.

Entretanto, nossas pesquisas identificaram que apenas a utilização do *Titan,* de forma distribuída, em conjunto com o *Cassandra* não seria a melhor escolha para atender as necessidades de processamento em paralelo da nossa proposta. Portanto, o *Faunus* ideal é uma solução mais adequada, por ser capaz de reestruturar o grafo massivo do *Titan*, realizando a quebra e distribuição do mesmo em sub grafos mais simples, para então distribuí-los pelas diversas máquinas que estarão na nuvem, e assim aumentar a capacidade de processamento do nosso sistema.

Resumidamente, a nossa proposta de arquitetura funciona da seguinte maneira, descrita na Figura 5.

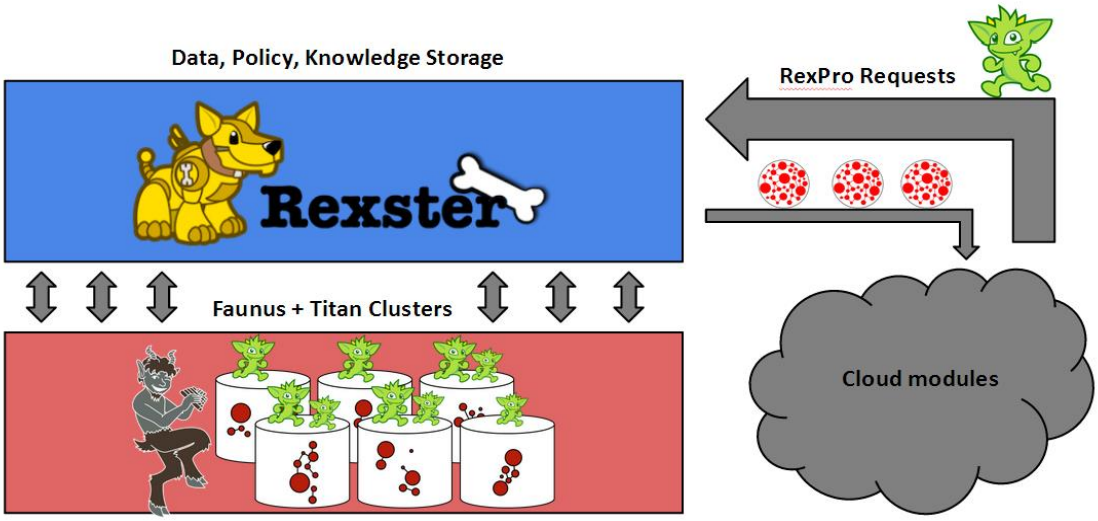

Figura 5 - Esquema da Arquitetura

Inicialmente outros módulos em *Cloud* irão interagir com a nossa arquitetura, estes módulos podem ser servidores de dados que fornecem informações provenientes de dispositivos espalhados em uma cidade, ou podem ser módulos de inteligências que utilizam os dados armazenados para um processamento mais específico.

Esses módulos se comunicariam com o servidor *Rexster*, podendo ser através da interface web "The Dog House" ou através do protocolo binário RexPro, o desenvolvedor do módulo pode optar pela melhor escolha.

Deste ponto em diante, serão enviados requisições em Gremlin para o servidor do *Rexster*, que por sua vez irá gerenciar e repassar as requisições para o Faunus. Este em conjunto com uma implementação do Titan utilizará como banco de dados no seu *backend* o Cassandra.

As requisições em Gremlin, serão então distribuídas através do *cluster* usando o Faunus, responsável por gerenciar tanto a dispersão do grafo do Titan quanto a execução das requisições em Gremlin, empregando a tecnologia Apache *Hadoop*.

Após a requisição ser processada, o Faunus encaminha o resultado para o servidor *Rexster*, estee por sua vez retornará apenas um resultado ou vários sub-grafos paraos módulos requisitantes.

Com esta arquitetura, observa-se que conseguimos atender a grande parte das características que envolvem o nosso problema, desde capacidade de processamento distribuído até armazenamento eficiente em colaboração com outros componentes da solução.

X. Implementação

Para validar a nossa proposta foi realizada uma implementação parcial voltada para aplicação em redes de sensores. O cenário proposto contém um número indeterminado de áreas, que por sua vez podem incluir ou não subáreas. Cada área ou sub-área poderá ter um número indefinido de dispositivos pertencentes a uma rede de sensores. Esses dispositivos irão realizar várias medições de diversos parâmetros ao longo do dia, sendo necessário o armazenamento do histórico destas medições, durante um intervalo de tempo indeterminado. O cenário deverá armazenar informações de identificação, como nome da área, endereço de identificação dos dispositivos, qual unidade onde estão sendo realizadas as medidas, e em qual tempo foi realizado determinada medida (*Time Stamp*).

Utilizando este cenário foi desenvolvida uma modelagem relacional para o banco de dados em grafos.[2]

A modelagem proposta visa reduzir ao máximo a quantidade de atributos pertencentes a cada nó, para permitir que as operações complexas percorram um nó com muitos atributos sem prejudicar o desempenho. Portanto, apenas os atributos estritamente necessários para identificação, e outras informações relevantes, como o tempo para a realização de uma determinada medida. Foi avaliado como mais vantajoso que as arestas representassem os relacionamentos do grafo. Neste contexto, foram criados nós pertencentes a um conjunto finito que representam o que está sendo medido por algum dispositivo, por exemplo, nós denominados, Humidade, Energia, Luminosidade, Temperatura e etc. O objetivo da criação destes nós foi remover um atributo extra em cada nó do tipo Dispositivo, criando apenas um relacionamento entre o nó do que está sendo medido com os dispositivos que estão utilizando aquele tipo de medida. Note que o relacionamento ocorre na direção do nó de Tipo para o nó Dispositivo, intuitivamente temos a idéia de realizar o relacionamento contrário. O motivo para realizarmos a modelagem desta maneira foi aumentar a performance do banco de dados quando da realização de operações

[2] Apesar deste tipo de banco de dados armazenarem dados não estruturados, é interessante criarmos uma estrutura de relacionamento, a fim de facilitar na busca e processamento dos dados.

transversais. Por exemplo, para consultar os dispositivos que medem o Tipo Energia, é necessário verificar apenas quais as arestas que saem deste tipo de nó, simplificando bastante a operação.

Outro problema abordado na modelagem foi a necessidade de arquivar o histórico de medições de cada dispositivo, de maneira a reduzir ao máximo o impacto destes dados na performance do sistema. Portanto o histórico foi modelado em formato de lista, onde cada Dispositivo contém seu próprio histórico, e a cada nova medida as antigas são inseridas após a nova, com isso apenas a medição mais recente contém um relacionamento direto com o Dispositivo. Esta solução aumenta a eficiência das solicitações que necessitam apenas da última medição realizada (mais usual), sem afetar buscas transversais no restante do grafo, pois uma busca só terá necessidade de passar pelo histórico de um Dispositivo caso seja solicitada explicitamente.

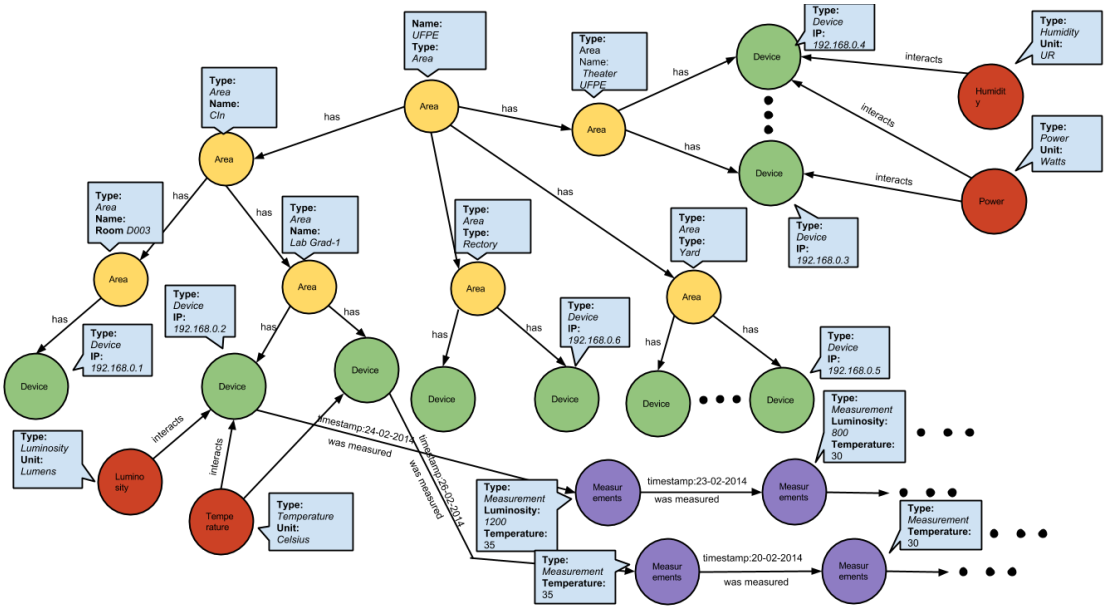

Figura 6 - Modelagem em grafo dos dados

Para a implementação desta modelagem descrita na Figura 6 foi utilizada a biblioteca BulbFlow [53], também conhecida como Bulbs.

Bulbs é um framework persistente em Python 3 e Python 2.7, responsável por conectar diversos tipos de banco dados baseados em grafos, como por exemplo o Neo4j ou o Stack Blueprints, permitindo acesso aos módulos e principalmente ao Servidor *Rexster* e consequentemente os banco

de dados suportados pelo mesmo. Uma das vantagens interessantes do Bulbs, é poder ser utilizado no desenvolvimento de aplicações para desktop como também em diversos frameworks de Python para Web, como por exemplo, o Django, Flask e Pyramid.

A. Avaliação

Para avaliar o desempenho da solução foi criado um banco de dados utilizando a modelagem descrita anteriormente, e um *script* desenvolvido em *Python* com a biblioteca *Bulbs*. O banco de dados foi preenchido da seguinte maneira:

- Total de Dispositivos Adicionados: 30
- Total de medições realizadas por cada dispositivo: 500
- Total de Áreas: 7
- Total de Tipos de Medições: 4

Com o objetivo de obter um banco de dados com informações próximas de uma execução real, foi adicionada sete áreas, contendo subáreas ou não. As áreas representam localizações de uma universidade, por exemplo. Quatro tipos diferentes de medidas (Energia, Temperatura, Humidade e Luminosidade) as mais utilizadas nas redes de sensores foram incluídas, e foram adicionados 30 dispositivos o que equivale a três dispositivo para cada área. Cada dispositivo realiza uma medição a cada 3 minutos durante um dia todo, totalizando aproximadamente 500 medições, por dispositivos diariamente.

Desta forma, foi gerado um grafo contendo 120.252 vértices e 120.966 arestas. A partir deste banco de dados foram realizadas diversas buscas para avaliar a performance do sistema. Os testes foram realizados em apenas 1 máquina com as seguintes configurações:

- Processador Intel (R) Core(TM) I3-2100 CPU @@ 3.10 Ghz
- Com 2 GB de Memória RAM.

O cenário dos testes executou o *Titan* contendo o banco de dados criado acima, juntamente com um servidor *Rexster*, foi executado um *script* em *Python* para realizar consultas *Gremlins* ao servidor *Rexster*. Foram realizadas

10.000 consultas de dispositivos aleatórios, sendo o tempo gasto aproximadamente de 200 segundos.

Nas Figuras 7, 8 e 9, podemos observar o comportamento dos dados através dos histogramas em diferentes escalas e partições, demonstrando que a grande das consultas estão foram realizadas entre 15 e 20 milissegundos.

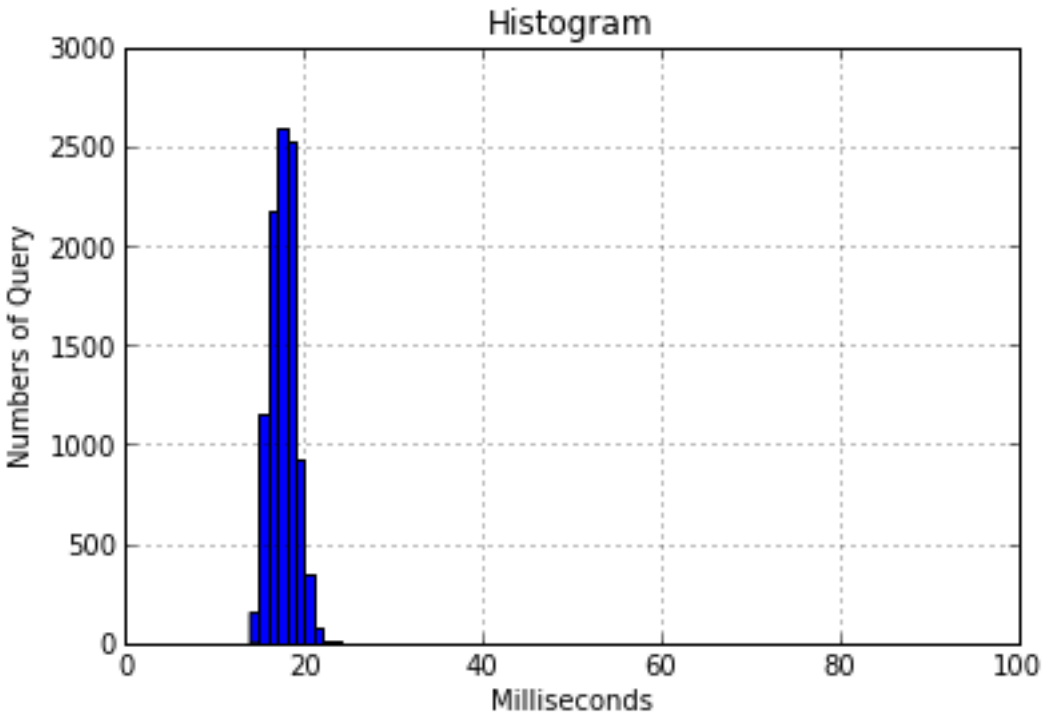

Figura 7 - Histograma - Consultas por Tempo de execução

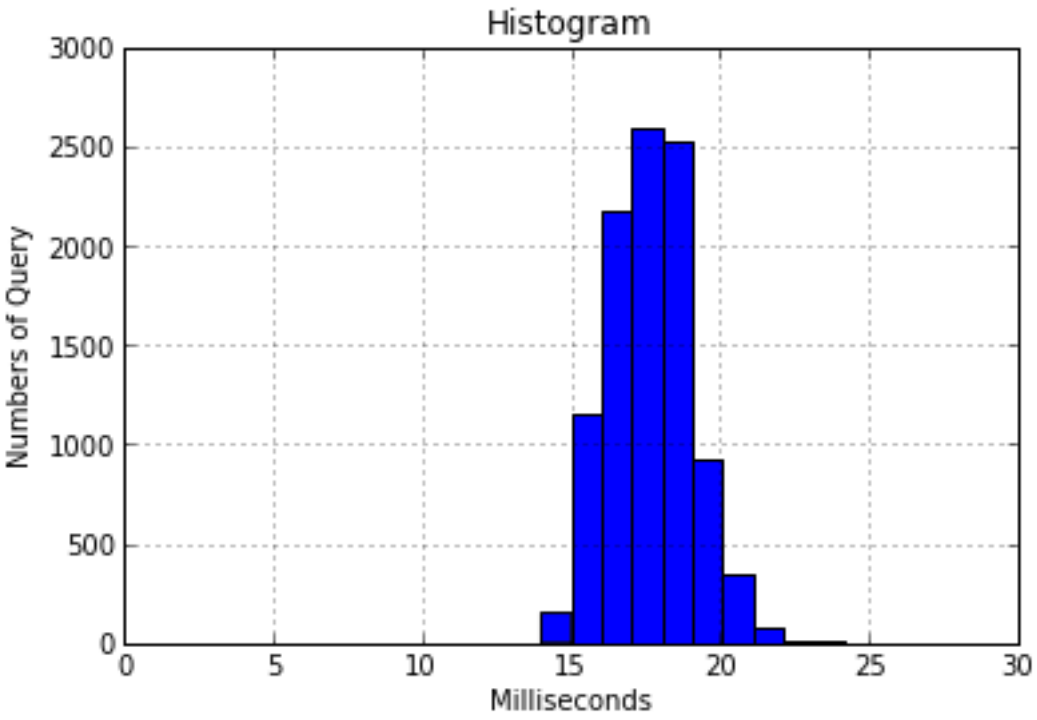

Figura 8 - Histograma - Consultas por Tempo de execução

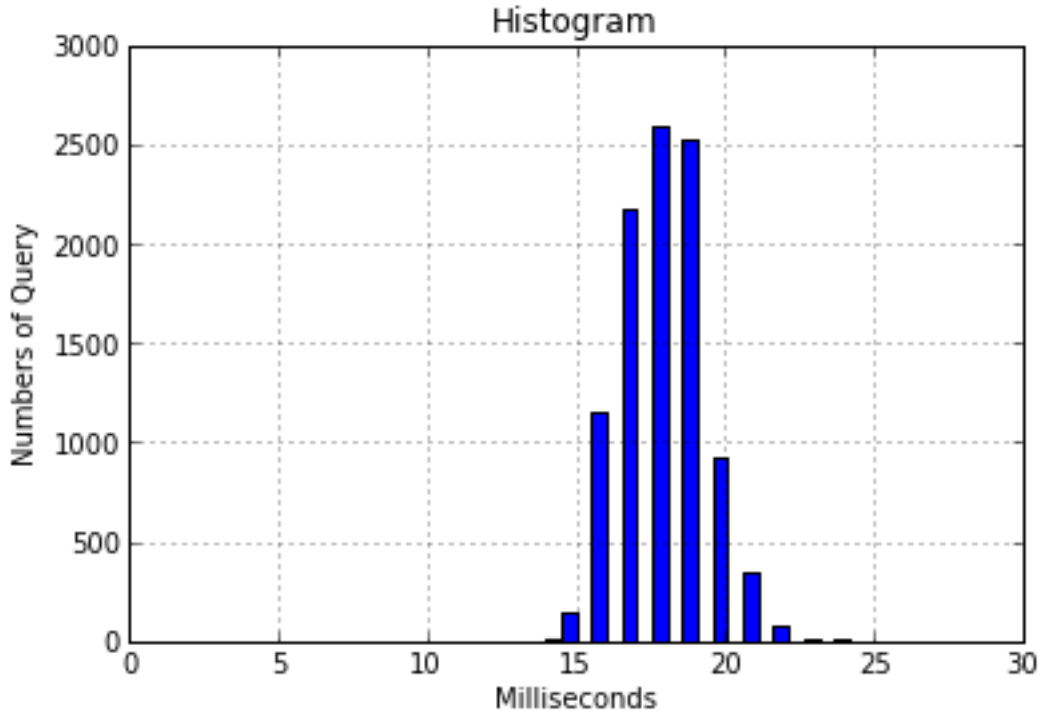

Figura 9 - Histograma - Consultas por Tempo de execução

Na Figura 10 e 11, observamos o gráfico dos tempos que as consultas levaram para serem executadas com sucesso, notamos que entre a consulta inicial (Ponto 0 no eixo x) e até próximo a consulta de número 140, ocorreram valores diferentes quando comparados com o resto do conjunto de dados, supostamente o sistema ainda estava realizando alguma operação interna que afetou a análise precisa destes valores iniciais, consideraremos então estes valores como *outliers* na nossa análise, afim de obtermos estatísticas mais reais, e assim assumimos que em um sistema real, o mesmo permanece em funcionamento constante por um certo período, sendo irrelevante estas diferenças observadas nas consultas iniciais.

Analisamos assim o gráfico a partir da consulta de número 140 até a última de número 10.0000.

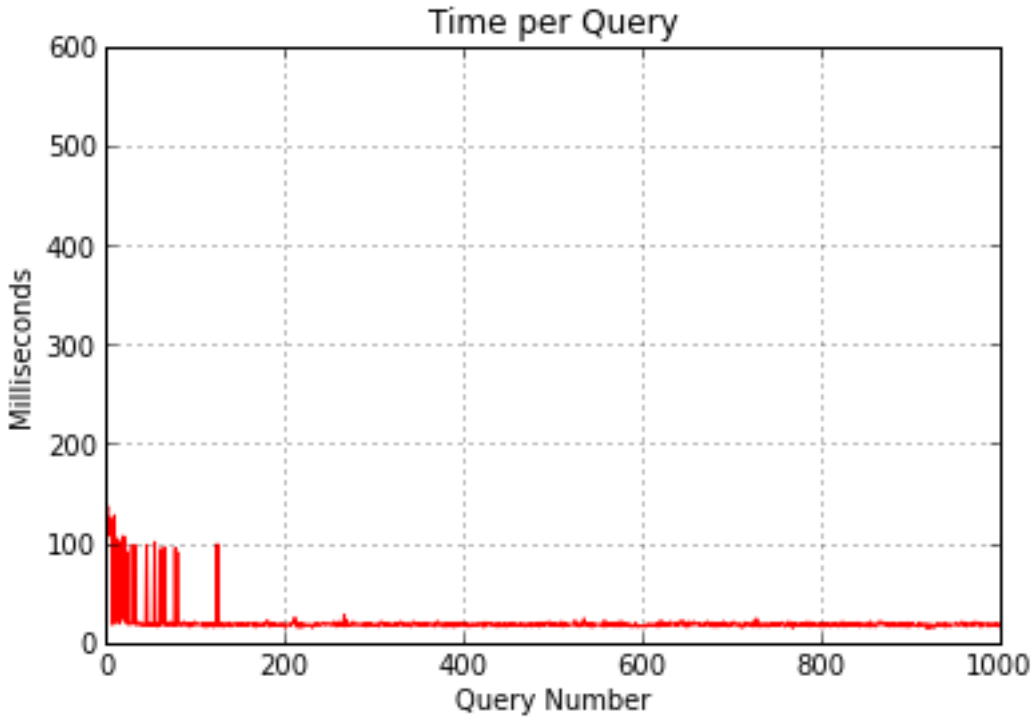

Figura 10 - Gráfico - Consultas por Tempo de execução

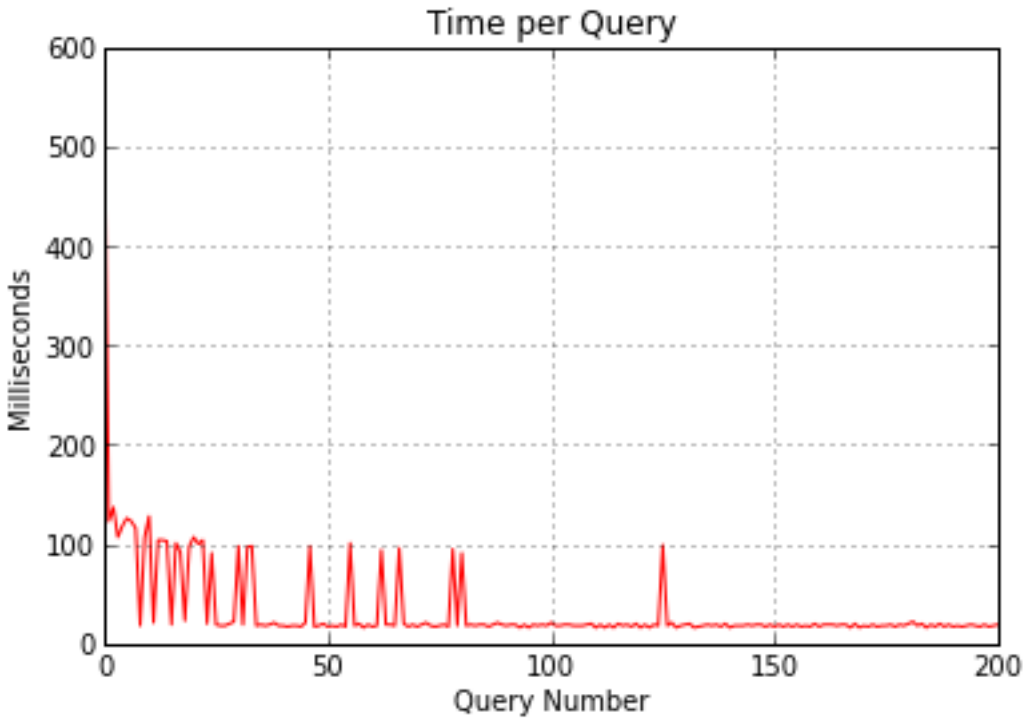

Figura 11 - Gráfico - Consultas por Tempo de execução

Nas Figuras 12 e 13, notamos um comportamento mais padrão das consultas, apesar das mesmas serem realizadas em busca de dispositivos aleatórios, nota-se que os valores de resposta permaneceram em torno de 20 milisegundos durante todas as consultas.

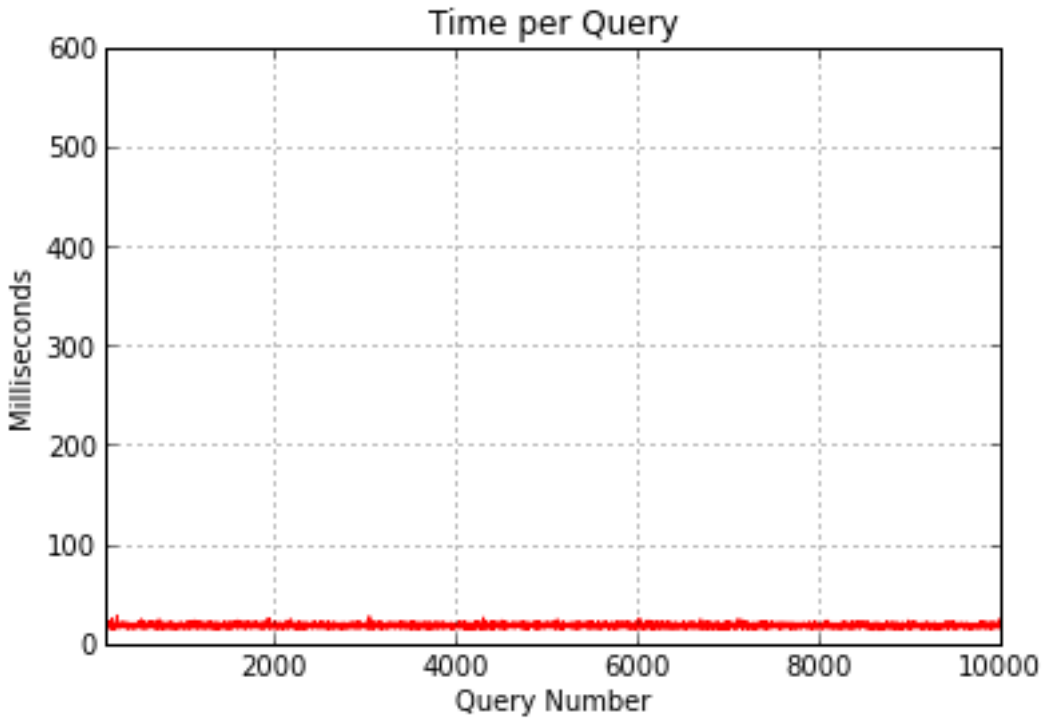

Figura 12 - Gráfico - Consultas por Tempo de execução

Figura 13 - Gráfico - Consultas por Tempo de execução

O tempo médio para cada consulta foi de 20 milissegundos. Vale salientar também que 75% das consultas foram executadas com tempo inferior

ao tempo médio. As estatísticas obtidas com este conjunto de dados estão descritas a seguir:

Total de Consultas: 10.000

Média do Tempo: 18.39 milissegundos

Desvio Padrão: 7.055

Menor Tempo: 14 milissegundos

Maior Tempo: 522 milissegundos

25% das consultas foram executadas em torno de 14 milissegundos

50% das consultas foram executadas em torno de 17 milissegundos

75% das consultas foram executadas em torno de 19 milissegundos

XI. Linguagem Específica do Domínio

Gremlin é uma linguagem bastante útil para a nossa proposta, entretanto a leitura de alguns códigos descritos em *Gremlin* pode ser uma tarefa difícil.

Por exemplo, realizar a consulta de apenas um dispositivo dado o seu ip, pode ser simples, como o comando abaixo:

g.V. *has('Type','Device').has('IP','192.168.0.1')*

Mas códigos complexos podem dificultar o entendimento, como o seguinte:

g.V.ifThenElse{ip==null}{it.has('Type','Device').has('IP',it.IP).in('has')}{it.has('Type','Device').has('IP',ip).in('has')}}

Para a nossa proposta ser viável para usuários que não detenham conhecimento destas novas tecnologias, seria necessário realizarmos a criação de uma linguagem específica do domínio (*Domain Specific Language – DSL*). Gremlin oferece suporte a criação de passos compostos (*Pipes*) utilizando os *Pipes* existentes ou criando Pipes novos. Desta forma foi realizada uma expansão da linguagem *Gremlin*, , ocultando a programação de baixo nível e permitindo a utilização do sistema sem conhecimento sobre banco de dados baseado em grafos.

Com a criação de um DSL permite-se a criação de funções de alto nível, por exemplo. pode-se realizar buscas específicas sobre dispositivos que medem apenas Energia, ou todos os Dispositivos de uma Área de maneira simplificada e eficiente, se comparado com o *Gremlin* puro e principalmente com o clásscio SQL.

A criação de um passo composto em Gremlin ocorre da seguinte maneira:

Precisamos definir os sequintes parêmetros:

- *StepName:* Nome que representará o passo composto.
- *List<Class>:* A lista de classes que serão suportadas
- *StepClosure*: O corpo do método

De posse destes parâmetros, um passo composto utilizando a função nativa do Gremlin será criado:

- *Gremlin.defineStep(String stepName, List<Class> classes, Closure stepClosure);*

Os passos compostos criados durante este TG estão descritos a seguir, uma descrição mais detalhada pode ser encontrada na seção “Apêndice”.

Passos Compostos Criados

Tabela 6 - Passos compostos implementados

Step	**description**	**return**
GET_AREA(String name)	Return object area if name is specified or a list of all areas if none name is passed	set of areas
GET_DEVICE(String ip)	Return a device if the ip is specified or return a list of all devices if none ip is passed.	set of devices
GET_AREA_PER_DEVICE(ip)	Return an area containing the device specified	set of areas
GET_AREA_PER_UNIT(unit)	Returns a list of area that contains the unit specified measured by at least one device.	set of areas

GET_DEVICE_PER_AREA(name)	Return a list of devices which is in the area.	set of devices
GET_DEVICE_PER_UNIT(type)	Return a list of devices which are measuring the unit passed	set of devices
GET_MEASUREMENT_FROM_AREA(name)	The return is all measurements from that area	set of measurement
GET_MEASUREMENT_FROM_DEVICE(ip)	The return is the last measurement from that ip	set of measurement
GET_MEASUREMENT_FROM_UNIT(type)	The return is all measurements of that unit type	set of measurement
GET_UNIT_PER_DEVICE(ip)	Show all units measured by the device specified on ip.	set of units
GET_UNIT_PER_AREA(name)	Show all units	set of units

	measured on the area which name is given.	

Passos Nativos do Gremlin

Tabela 7 - Passos Nativos do Gremlin

Step	**description**	**return**
Map	Show all the property and values from that element. Eg: data.map	description of the element
Next	Get the next element from a set. Next can be applied so many as you can. Eg: set.next().next().next() Will return the 3rd element from set.	next element of the set
.property*	Access the property value from some element or relationship. Show: data.property* Update: data.property* = new_value Where property* is the name of property.	
.remove()	Remove the element from database.	

	Eg: data.remove() data element now is removed	

De forma complementar aos Passos Compostos, temos a noção de funções, o comportamento das funções é diferente dos passos compostos, as memas não são aninhadas na forma de *Pipes*, mas sim executadas retornando uma saída. Funções foram utilizadas nesta proposta para a criação de operações mais complexas, como a criação de algum vértice, pois envolvem mais de uma operação e necessitam de variáveis internas para processamento, sendo assim inviável o uso de Pipes para o seu desenvolvimento.

Tabela 9 - Funções Implementadas

Function	**description**	**result**
CREATE_DEVICE(ip, area_contained,measure_type)	Create and save on database a new device and a relationship between the new device and area e types of measurements	The representation of relationship, if the device was sucessfull added, or error message if an error ocurred.
CREATE_AREA(name,area_contained)	Create and save on database a new area and a relationship between the new area and area which is included.	The representation of relationship, if the device was sucessfull added, or error message if an error ocurred.

XII. Conclusões e Trabalhos Futuros

A tecnologia de *CloudCloud Computing* é quase mandatória para o tratamento de *Big Data* proveniente não só da *Internet Of Things*, mas de diversas fontes. Em relação ao banco de dados baseados em grafos observa-se que o relacionamento entre os dados é tão importante e rico de informações quanto os dados em sí, permitindo que sejam realizadas análises não apenas estatísticas, mas também semânticas analisando apenas a estrutura do grafo, tendo em vista que os dados estão interligados entre sí.

Com a junção das vantagens dos bancos de dados em grafo com tecnologias de *Cloud Computing*, é possível alcançar de forma satisfatória os requisitos de elasticidade de recursos e escalabilidade tanto de processamento quanto de armazenamento, com maior eficiência e envolvendo o menor custo possível (Engenharia), como demonstrado em [44] e demais análises ao longo deste documento.

A solução da *Aurelius, Faunus* e o *Blueprints* da *Tinkerpop*, tornou o sistema mais maleável e adaptável a diversas características, demonstrando assim que a nossa proposta pode ser útil não apenas no contexto de *Internet Of Things*, mas está aberta para propósito geral que envolva *Big Data* em geral ou características da mesma.

Como sugestão para trabalhos futuros, estender as avaliações realizadas neste TG para um ambiente mais próximo de uma real utilização, por exemplo, utilizando uma infraestrutura em *Cloud* com uma carga semelhante à encontrada em uma aplicação. Outra expansão seria considerar diversos domínios, extrapolado o escopo de redes de sensores utilizado como exemplo, e abordando domínios maiores ou mais específicos de outras áreas

XIII. Referências

[1] A. Zaslavsky, C. Perera, and D. Georgakopoulos, “Sensing as a service and big data,” … *Prepr. arXiv1301.0159*, 2013.

[2] D. Evans, “The Internet of Things How the Next Evolution of the Internet The Internet of Things How the Next Evolution of the Internet Is Changing Everything,” no. April, 2011.

[3] “Welcome to Apache™ Hadoop®!” [Online]. Available: http://hadoop.apache.org/. [Accessed: 26-Feb-2014].

[4] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, 2008.

[5] C. Strauch, U. Sites, and W. Kriha, “NoSQL databases,” *URL http//www. christof-strauch. de/ …*, 2011.

[6] “The World’s Smartest Cities - Forbes.” [Online]. Available: http://www.forbes.com/2009/12/03/infrastructure-economy-urban-opinions-columnists-smart-cities-09-joel-kotkin.html. [Accessed: 20-Feb-2014].

[7] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” *Futur. Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.

[8] M. Cherniack, H. Balakrishnan, and M. Balazinska, “Scalable Distributed Stream Processing.,” *CIDR*, 2003.

[9] A. Buchmann and B. Koldehofe, “Complex event processing,” *it-Information …*, 2009.

[10] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud computing and grid computing 360-degree compared,” *Grid Comput. Environ. …*, 2008.

[11] W. Gropp, E. Lusk, and A. Skjellum, “Using MPI: portable parallel programming with the message-passing interface,” 1999.

[12] C. Pettey and R. van der Meulen, “Gartner’s 2012 hype cycle for emerging technologies identifies ‘Tipping Point’ technologies that will unlock long-awaited technology scenarios: Gartner,” 2012.

[13] S. Sakr and A. Liu, “SLA-based and consumer-centric dynamic provisioning for cloud databases,” *Cloud Comput. (CLOUD), 2012 IEEE 5th …*, 2012.

[14] “Amazon Web Services (Portuguese (Brazil)).” [Online]. Available: http://aws.amazon.com/pt/. [Accessed: 26-Feb-2014].

[15] “OpenNebula | Flexible Enterprise Cloud Made Simple.” [Online]. Available: http://opennebula.org/. [Accessed: 26-Feb-2014].

[16] I. Mapanga and P. Kadebu, “Database Management Systems : A NoSQL Analysis,” no. 7, 2013.

[17] “Riak | Basho Technologies.” [Online]. Available: http://basho.com/riak/. [Accessed: 10-Mar-2014].

[18] “MongoDB.” [Online]. Available: http://www.mongodb.org/. [Accessed: 10-Mar-2014].

[19] “Apache CouchDB.” [Online]. Available: http://couchdb.apache.org/. [Accessed: 10-Mar-2014].

[20] “The NoSQL database software that delivers instant full-text search and SQL-like query across mixed data types, database ranking for superior relevancy and native scale out ability.” [Online]. Available: http://www.clusterpoint.com/. [Accessed: 10-Mar-2014].

[21] O. Lassila and R. Swick, “Resource Description Framework (RDF) model and syntax specification,” 1998.

[22] L. Antsfeld and T. Walsh, “Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm,” *… Transp. Syst. World Congr.*, 2012.

[23] D. Ritter, “From network mining to large scale business networks,” *Proc. 21st Int. Conf. …*, 2012.

[24] L. Royer and M. Reimann, “Unraveling protein networks with power graph analysis,” *PLoS Comput. …*, 2008.

[25] C. Bizer, J. Lehmann, and G. Kobilarov, “DBpedia-A crystallization point for the Web of Data,” *… World Wide Web*, 2009.

[26] C. Vicknair, M. Macias, Z. Zhao, and X. Nan, “A comparison of a graph database and a relational database: a data provenance perspective,” *Proc. 48th …*, 2010.

[27] R. Angles, “A comparison of current graph database models,” *Data Eng. Work. (ICDEW), 2012 IEEE …*, 2012.

[28] M. Ciglan, A. Averbuch, and L. Hluchy, “Benchmarking traversal operations over graph databases,” *Data Eng. Work. …*, 2012.

[29] D. Dominguez-Sal, “Survey of graph database performance on the hpc scalable graph analysis benchmark,” *Web-Age Inf. …*, 2010.

[30] “TinkerPop.” [Online]. Available: http://www.tinkerpop.com/. [Accessed: 20-Feb-2014].

[31] “Neo4j - The World’s Leading Graph Database.” [Online]. Available: http://www.neo4j.org/. [Accessed: 10-Mar-2014].

[32] “Titan: Distributed Graph Database.” [Online]. Available: http://thinkaurelius.github.io/titan/. [Accessed: 10-Mar-2014].

[33] “Semantic Technologies AllegroGraph Triple Store RDF Web 3.0 Database,optimized SPARQL Query engine, Prolog and RDFS+ reasoner.” [Online]. Available: http://franz.com/agraph/allegrograph/. [Accessed: 10-Mar-2014].

[34] “Sparsity-technologies: Sparksee high-performance graph database.” [Online]. Available: http://sparsity-technologies.com/. [Accessed: 10-Mar-2014].

[35] S. Jouili and V. Vansteenberghe, “An Empirical Comparison of Graph Databases,” *2013 Int. Conf. Soc. Comput.*, pp. 708–715, Sep. 2013.

[36] S. Matsuoka, “TSUBAME 2.0 Begins,” *sim.gsic.titech.ac.jp*.

[37] “Aurelius.” [Online]. Available: http://thinkaurelius.com/. [Accessed: 20-Feb-2014].

[38] “NuvolaBase: cloudize your data - Commercial support, training and services about OrientDB.” [Online]. Available: http://www.nuvolabase.com/site/. [Accessed: 10-Mar-2014].

[39] “CloudGraph .NET graph database.” [Online]. Available: http://www.cloudgraph.com/. [Accessed: 10-Mar-2014].

[40] “Heroku | Cloud Application Platform.” [Online]. Available: https://www.heroku.com/new. [Accessed: 10-Mar-2014].

[41] S. Banon, “Elasticsearch: An open source, distributed, RESTful search engine,” 2011.

[42] “Apache Lucene - Welcome to Apache Lucene.” [Online]. Available: https://lucene.apache.org/. [Accessed: 10-Mar-2014].

[43] J. Han, E. Haihong, G. Le, and J. Du, “Survey on NoSQL database,” *Pervasive Comput. …*, 2011.

[44] “Titan Provides Real-Time Big Graph Data | Aurelius on WordPress.com.” [Online]. Available: http://thinkaurelius.com/2012/08/06/titan-provides-real-time-big-graph-data/. [Accessed: 10-Mar-2014].

[45] A. Lakshman and P. Malik, “Cassandra: structured storage system on a p2p network,” *Proc. 28th ACM Symp. …*, 2009.

[46] “AWS | Amazon DynamoDB – NoSQL Database Service.” [Online]. Available: http://aws.amazon.com/dynamodb/. [Accessed: 20-Feb-2014].

[47] F. Chang, J. Dean, and S. Ghemawat, “Bigtable: A distributed storage system for structured data,” *… Comput. Syst. (…*, 2008.

[48] E. Dede, B. Sendir, P. Kuzlu, J. Hartog, and M. Govindaraju, “An Evaluation of Cassandra for Hadoop,” *2013 IEEE Sixth Int. Conf. Cloud Comput.*, pp. 494–501, Jun. 2013.

[49] “Rexster - Tinkerpop.” [Online]. Available: http://rexster.tinkerpop.com/. [Accessed: 10-Mar-2014].

[50] “Pipes - Tinkerpop.” [Online]. Available: http://pipes.tinkerpop.com/. [Accessed: 10-Mar-2014].

[51] F. Holzschuher, “Performance of Graph Query Languages Comparison of Cypher , Gremlin and Native Access in Neo4j,” pp. 195–204, 2013.

[52] “Faunus: Graph Analytics Engine.” [Online]. Available: http://thinkaurelius.github.io/faunus/. [Accessed: 10-Mar-2014].

[53] “Bulbflow: a Python Framework for the Graph Era.” [Online]. Available: http://bulbflow.com/. [Accessed: 20-Feb-2014].

XIV. Apêndice

Código dos passos compostos e funções implementados em *Gremlin*

GET_AREA(name)
Arguments:
name: String , null

Return:
Return object area if name is specified or a list of all areas if none name is passed

Definition:

```
Gremlin.defineStep('GET_AREA',[Vertex,Pipe], {name ->  _().ifThenElse{name == null}{it.has('Type','Area')}{it.has('Type','Area').has('Name',name)}})
```

GET_AREA_PER_DEVICE(ip)

Arguments:
ip: String

Return:
Return an area containing the device especified

Definition:

```
Gremlin.defineStep('GET_AREA_PER_DEVICE',[Vertex,Pipe], {ip ->  _().ifThenElse{ip == null}{it.has('Type','Device').has('IP',it.IP).in('has')}{it.has('Type','Device').has('IP',ip).in('has')}})
```

GET_AREA_PER_UNIT(unit)
Arguments:
unit: String

Return:

Returns a list of area that contains the unit specified measured by at least one device.

Definition:

```
Gremlin.defineStep('GET_AREA_PER_UNIT',[Vertex,Pipe], {unit -
> _().ifThenElse{unit ==
null}{it.has('Type',it.Type).out.GET_AREA_PER_DEVICE}{it.has('Type',unit).out
.GET_AREA_PER_DEVICE}})
```

GET_DEVICE(ip)

Arguments:

ip: String, null

Return:

Return a device if the ip is specified or return a list of all devices if none ip is passed.

Definition:

```
Gremlin.defineStep('GET_DEVICE',[Vertex,Pipe], {ip -> _().ifThenElse{ip ==
null}{it.has('Type','Device')}{it.has('Type','Device').has('IP',ip)}})
```

GET_DEVICE_PER_AREA(name)

Arguments:

name: String

Return:

Return a list of devices which is in the area.

Definition:

```
Gremlin.defineStep('GET_DEVICE_PER_AREA',[Vertex,Pipe], {name -
> _().ifThenElse{name ==
null}{it.has('Type','Area').has('Name',it.Name).out.has('Type','Device')}{it.has('Ty
pe','Area').has('Name',name).out.has('Type','Device')}})
```

GET_DEVICE_PER_UNIT(type)

Arguments:

type: String

Return:

Return a list of devices which are measuring the unit passed

Definition:

```
Gremlin.defineStep('GET_DEVICE_PER_UNIT',[Vertex,Pipe], {unit -
>  _().ifThenElse{unit == null}{it.has('Type',it.Type).out}{it.has('Type',unit).out}})
```

GET_MEASUREMENT_FROM_AREA(name):

Arguments:

name: String, null

Return:

The return is all measurements from that area

Definition:

```
Gremlin.defineStep('GET_MEASUREMENT_FROM_AREA',[Vertex,Pipe],
{name ->  _().ifThenElse{name ==
null}{it.has('Type','Area').has('Name',it.Name).out.has('Type','Device').out('was
measured')}{it.has('Type','Area').has('Name',name).out.has('Type','Device').out('
was measured')}})
```

GET_MEASUREMENT_FROM_DEVICE(ip)

Arguments:

ip: String

Return:

The return is the last measurement from that ip

Definition:

```
Gremlin.defineStep('GET_MEASUREMENT_FROM_DEVICE',[Vertex,Pipe], {ip
->  _().ifThenElse{ip == null}{it.has('Type','Device').has('IP',it.IP).out('was
measured')}{it.has('Type','Device').has('IP',ip).out('was measured')}})
```

GET_MEASUREMENT_FROM_UNIT(type)

Arguments:

type: String

Return:

The return is all measurements of that unit type

Definition:

```
Gremlin.defineStep('GET_MEASUREMENT_FROM_UNIT',[Vertex,Pipe], {type
->  _().ifThenElse{type ==
null}{it.has('Type',it.Type).GET_DEVICE_PER_UNIT.GET_MEASUREMENT_F
ROM_DEVICE}{it.has('Type',type).GET_DEVICE_PER_UNIT.GET_MEASURE
MENT_FROM_DEVICE}})
```

CREATE_DEVICE(ip, area_contained,measure_type)

Arguments:

ip, area_contained: String

measure_type: List<type>

Return:

True, if the device was sucessfull added, or False if an error ocurred.

Definition:

```
def CREATE_DEVICE(ip, area_contained ,measure_type)
{
        device = g.addVertex([Type:'Device',IP:ip]);
        g.addEdge(area_contained,device, 'has');
        g.addEdge(measure_type,device,'interacts')
}
```

CREATE_AREA(name,area_contained)
Arguments:
name, area_contained: String

Return:
True, if the are was sucessfull added, or False if an error ocurred.

Definition:

```
def CREATE_AREA(name,area_contained)
{
	area = g.addVertex([Type:'Area',Name:name]);
	g.addEdge(area_contained,area, 'has');
}
```

GET_UNIT_PER_DEVICE(ip)
Arguments:
ip: String

Return:
Show all units measured by the device specified on ip.

Definition:

```
Gremlin.defineStep('GET_UNIT_PER_DEVICE',[Vertex,Pipe], {ip -
>  _().ifThenElse{ip ==
null}{it.has('Type','Device').has('IP',it.IP).in('interacts')}{it.has('Type','Device').has
('IP',ip).in('interacts')}})
```

GET_UNIT_PER_AREA(name)
Arguments:
name: String

Return:
Show all units measured on the area which name is given.

Definition:

```
Gremlin.defineStep('GET_UNIT_PER_AREA',[Vertex,Pipe], {name -
> _().ifThenElse{name ==
null}{it.has('Type','Area').has('Name',it.Name).GET_DEVICE_PER_AREA.GET_
UNIT_PER_DEVICE}{it.has('Type','Area').has('Name',name).GET_DEVICE_PE
R_AREA.GET_UNIT_PER_DEVICE}})
```