UNIVERSIDADE FEDERAL
DE SANTA CATARINA

ARMAZENAMENTO BIG DATA NO MONITORAMENTO EM NUVEM

MAXWELL GONÇALVES DE ALMEIDA

Florianópolis-SC

2014/1

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

ARMAZENAMENTO BIG DATA NO MONITORAMENTO EM NUVEM

MAXWELL GONÇALVES DE ALMEIDA

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Carlos Becker Westphall
Coorientador: Prof. Ms. Rafael de Souza Mendes

Florianópolis-SC

2014/1

MAXWELL GONÇALVES DE ALMEIDA

ARMAZENAMENTO BIG DATA NO MONITORAMENTO EM NUVEM

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientador:

_____

Prof. Dr. Carlos Becker Westphall
Orientador

Banca Examinadora:

_____

Prof. Ms. Rafael de Souza Mendes
Coorientador

_____

Prof.ª Dr.ª Carla Merkle Westphall
Membro da Banca

_____

Bel. Fernando Schubert
Membro da Banca

Aos meus e à Deus.

AGRADECIMENTOS

Agradeço principalmente o Prof. Dr. Carlos Becker Westphall por ter me aberto a porta para o conhecimento desta área e a oportunidade de poder fazer o TCC sob sua orientação. Especial agradecimento ao meu co-orientador, Prof. Ms. Rafael de Souza Mendes pela sua dedicação, atenção, solicitude e sempre boa disposição a me ajudar nessa pequena jornada. Queria também agradecer os membros do *Laboratório de Redes e Gerência (LRG)*: [1] a Prof. Drª Carla Merkle Westphall, os mestrandos Fernando Schubert e Rafael Weingartner, e o graduando Gabriel Beims Bräscher.

Nunca poderia me esquecer dos meus pais que sempre me apoiaram, minha esposa Karolina Koszalska de Almeida, que acompanhou de perto essa fase final e meus amigos que mesmo distantes são sempre presentes.

[1] Laboratório de Redes e Gerência (**LRG**) <http://www.lrg.ufsc.br>

“Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que lhe é agradável, o que é perfeito.”
(Bíblia Sagrada, Romanos 12, 2)

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

AM	Agentes de Monitoramento
AMNCA	Arquitetura de Monitoramento em Nuvem Cognitivo-Autônoma
ARPANET	Advanced Research Projects Agency Network
BD	Banco de Dados
BDR	Banco de Dados Relacionais
AMQP	Advanced Message Queing Protocol
API	Application Programming Interface
BC	Bases de Conhecimento
CA	Computação Autônoma
CC	Computação Cognitiva
CG	Computação em Gride
CN	Computação em Nuvem
CNA	Computação em Nuvem Autônoma
CU	Computação Utilitária
CaaS	Communications as a Service
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language
DQL	Data Query Language
E/S	Entrada/Saída
FM	Ferramenta de Monitoramento

FW	Framework
GN	Gerenciador de Nuvem
HDFS	Hadoop Data File System
HVM	Hardware Virtual Machine
IDE	Integrated Development Environment
IaaS	Infrastructure as a Service
ITU-T	International Telecommunications Union - Telecommunication
LRG	Laboratório de Redes e Gerência
MAPE-K	Monitor, Analyse, Plan, Execute, and Knowledge
MD	Massa de Dados
MD	Modelo de Dados
MI	Monitoramento de Infraestrutura
MLD	Modelo Lógico de Dados
MM	Métricas de Monitoramento
MN	Monitoramento em Nuvem
MR	Monitoramento em Rede
MS	Monitoramento de um Serviço
MV	Máquina Virtual
MW	Middleware
NaaS	Network as a Service
NC	Nuvem Comunitária
NFS	Network File System
NIST	National Institute of Standards and Technology

NA	Nuvem Autônoma
NPi	Nuvem Privada
NH	Nuvem Híbrida
NPu	Nuvem Pública
PV	Para-Virtualização
PaaS	Platform as a Service
PD	Ponto de Dados
PU	Puramente Virtual
QoS	Qualidade de Serviço
RAID	Redundant Array of Independent Drives
RAM	Random Access Memory
RN	Regras de Negócio
REST	Representationl State Transfer
RoR	Ruby on Rails
SaaS	Software as a Service
SC	Sistemas Computacionais
SCSI	Small Computer System Interface
SD	Sistema Distribuido
SGC	Sistema de Gerenciamento de Configuração
SGDB	Sistema de Gerenciamento de Banco de Dados
SI	Sistemas de Informação
SO	Sistema Operacional
SOA	Service Oriented Architecture

SQL	Structured Query Language
SRT	Satisfação da Restrição Temporal
SSD	Sistema de Suporte a Decisão
SW	Software
ST	Séries Temporais
SV	Sistema de Virtualização
TD	Teoria da Decisão
TD	Tomada de Decisão
TI	Tecnologia de Informação
TM	Taxonomia do Monitoramento
TNST	Tabelas Não Seguras com Transação
TST	Tabelas Seguras com Transação
VMM	Virtual Machine Manager
VMM	Virtual Machine Monitor
VT	Virtualização Total

RESUMO

O cenário do monitoramento de redes está em constante mudança. Não faz muito tempo a evolução do paradigma de Grades para o de Nuvens gerou uma necessidade de adaptação das já consagradas ferramentas de monitoramento. No intento do redesenho da arquitetura de monitoramento de uma Nuvem foi planejada uma nova Arquitetura Cognitivo Autônoma de Monitoramento em Nuvem. Para tal, é preciso repensar quais ferramentas de monitoramento utilizar, bem como, de que maneira as métricas de uma Nuvem serão armazenadas para o monitoramento da mesma. Isto possibilita uma possível análise da frequência de eventos na Nuvem e, eventual tomada de decisão relativa aos recursos disponibilizados na mesma. Até alcançar tal nível de complexidade na implementação do projeto, é preciso particionar o problema e observar bem o cenário real das Nuvens com o rigor científico necessário, onde uma massa de dados monitorados é gerada devido a quantidade constante eventos registrados em uma *Cloud* de arquitetura complexa e com grande quantidade de máquinas físicas e virtuais. Portanto, não estamos mais falando na tecnologia tradicional de banco de dados relacional, já que atualmente existem vários sistemas de banco de dados adaptados ao paradigma *Big Data*. O foco deste trabalho é exatamente fazer um estudo que vai direcionar ao melhor modelo de dados para um esquema NoSQL do sistema HBase com o intuito de armazenar as métricas da nuvem do LRG, neste percurso é analisado o porquê da escolha do HBase num modelo de armazenamento em séries-temporais, o porquê de um modelo intervalar-temporal e o estudo de caso de um certo esquema de Banco de Dados adotado, bem como, uma comparação das vantagens e desvantagens do uso do NoSQL e não do tradicional SQL neste contexto através de um experimento com teste de unidade.

Palavras-chave: Big Data. Computação Autonômica. Computaçao Cognitiva. Computação em Nuvem. Esquema HBase. NoSQL. SQL. Teoria da Decisão.

ABSTRACT

The networks monitoring scenario is constantly changing. Not so long time ago, the evolution from grids paradigm to clouds generated a need to notch already established monitoring tools. Therefore, right now was designed a new Cognitive-Autonomic Monitoring Architecture for Clouds including new paradigms to match the real needs of that new pattern. To place that, we need to think which monitoring tools to deploy, as well as, how the metrics of a Cloud will be stored for its monitoring, being either in real time or whether in a longer timeline, eventually a time series. This provides a possible analysis of frequency in a sort of events at a Cloud, and eventual decision making concerning to find out resources available on it. To reach such a level of complexity in project implementation is necessary to break the problem and review a real scenario of clouds with scientific accuracy, where a mass of monitored data generated due to the massive amount of events, eventually from a complex architecture and large amount of physical and virtual machines as data source. Thus, we are not talking about traditional relational databases technology, since currently there are several database systems appropriate to Big Data paradigm. The focus of this final work is exactly do a study driven to release a better data model and schema of NoSQL HBase in place of store the monitored metrics of LRG's Cloud. Along the way, is analyzed choosing of HBase as interval-based temporal time-series database system, a case study within pros and cons of a certain schema, as well as, a comparison of NoSQL versus SQL through an unit test experiment.

Key-words: Autonomic Computing. Big Data. Cognitive Computing. Cloud Computing. Decision Theory. NoSQL. HBase Schema. SQL.

SUMÁRIO

1 INTRODUÇÃO

A evolução das *tecnologias de informação (TI)* possibilitaram diversos tipos de serviços para os seus usuários na última década por intermédio de uma das tecnologias mais recentes, a computação em nuvem (CN). As expressões *cloud computing (CC)* e *big data*[1] são por vezes tão citados hoje em dia entre os usuários assíduos de tecnologia, que muitas pessoas tratam como uma *buzzword*[2], (BESS, 2011). Um dos serviços que mais popularizou o aparecimento da *"nuvem virtual"* na vida das pessoas foi o *disco virtuai*, assim como para *a massa de dados (big data)*, a popularização do conceito se deu devido a grande quantidade de informação gerada na Web e tratada pelos mais variados serviços de busca e redes sociais. A *CN* também serve de vitrine de outras tecnologias em consolidação como o *big data*, incluindo também algumas já consolidadas e outras que sofreram adaptações para se alinharem com a realidade da arquitetura na Nuvem, são elas: *computação utilitária (CU)*, *computação em grade (CG)*, *Virtualização*, *computação autônoma (CA)* e *monitoramento em nuvem (MN)*.

Do conceito de *computação utilitária* surge a ideia de mercantilização de recursos computacionais e de armazenamento com a aquisição desses recursos sob demanda, e a possibilidade de desistir do serviço quando não for mais necessário, sem se preocupar com um orçamento significativo para *hardware* e gerenciamento.

Com o objetivo de obter uma melhor compreensão acerca dos pontos-chave envolvidos na implementação de serviços na Nuvem, na dificuldade associada ao monitoramento dos *ambientes em nuvem (AN)* e, uma falta de padronização até a atualidade, faz-se necessário uma análise cuidadosa e sistemática de um ambiente de operação heterogêneo e multicamadas. Tudo isso está relacionado à falta de maturidade típica das tecnologias recentes, o que se traduz em algumas limitações e questões ainda em âmbito de pesquisa. Segundo Iyer et al. (2009), a administração dos serviços em Nuvem é sustentada por ações como *visualização*, *controlabilidade* e *automação* de ambientes virtuais, essas ações são como se segue:

---

[1] Refere-se à um grande armazenamento de dados e maior velocidade nas transações de dados em um Sistema de Banco de Dados.

[2] Palavra utilizada mais para impressionar do que explicar o que realmente significa. <http://en.wikipedia.org/wiki/Buzzword>

Visualização: É uma vantagem para um administrador de serviços poder visualizar a gerência, na medida em que ajuda a responder rapidamente a eventos e a tomar melhores decisões;

Controlabilidade: Ajuda a gerir riscos e os custos de compartilhamento;

Automação- Ajuda a reduzir custos, trazendo agilidade às operações.

Além disso, pode-se afirmar que se nas *redes* convencionais ou *grades* o monitoramento era importante, na *detecção de problemas de rede*, *consumo de recursos* ou *disponibilidade dos serviços*, a importância se torna ainda maior na Nuvem, pois são mais variados e complexos. A *CN* possui modelos de serviço distintos mas que compartilham de pontos comuns, principalmente na *medição de infraestruturas*, tipo de serviço prestado, e recursos utilizados para os serviços.

Com o aumento da utilização da Internet e a proliferação de dispositivos habilitados para a web, serviços online, como o Gmail™, Facebook™, há um grande tráfego de dados, que é registrado em detalhes, resultando em *terabytes* até *petabytes* de dados. Esses registros fornecem uma visão de como os usuários estão utilizando o serviço e outras informações críticas de negócios tais como, fraude e detecção de anomalias. Os conjuntos de dados em tais dimensões devem ser distribuídos através de uma coleção de máquinas, e como os conjuntos de dados crescem, tecnologias relacionais existentes têm mostrado *grande atraso para cumprir consultas SQL* sofisticadas que requerem um novo conjunto de tecnologias integradas para grandes volumes de dados. Armazenamentos de dados *NoSQL* foram adotadas para lidar com esses conjuntos de dados, fornecendo *replicação*, *tolerância a falhas* e *alta disponibilidade*.

Conforme o que escreveu Mappic (2014, tradução nossa) em um artigo da internet muito recente: O termo *big data* é possivelmente um dos mais difíceis termos relacionados à *TI* de ser ignorado ultimamente. Existem tantos tipos de potenciais aplicações para *massa de dados* que pode ser um pouco assustador para considerar todas as possibilidades.

Os serviços *TI* distribuídos, isso inclui serviços em Nuvem, geram muitos dados além dos que são consequência da iteração dos usuários, contam também com um grande número de servidores, sejam eles virtuais ou físicos, bem como aplicativos e recursos de rede. Todos os sistemas, tecnologias e infraestrutura de *TI* geram dados relacionados com a

“saúde”, desempenho e estado dos serviços em execução em uma Nuvem, por exemplo. O volume de dados gerados por esta infraestrutura pode ser enorme, mas se correlacionada corretamente pode dar uma visão crítica sobre:

- O tempo e o comportamento do serviço ou resposta para uma requisição como no caso de um sistema de reservas aéreas;
- A causa de problemas de desempenho de um serviço,
- Análise de tendências e planejamento proativo estabelecidos em *Acordo de Nível de Serviço* (ANS).

Estes dados são analisados e processados em tempo real, a fim de dar uma resposta instantânea e de alerta para a degradação de qualquer serviço. Isto tudo se refere a parte de *monitoramento de um serviço (MS)*. Entretanto, essa *massa de dados* que está sendo coletada pode ser estruturada ou não, provenientes de uma variedade de sistemas que dependem uns dos outros para oferecer o melhor desempenho, e tem pouca ou nenhuma ligação óbvia entre eles, ou seja, os dados de uma aplicação são completamente independentes dos dados provenientes da rede executada. Quando estes dados advém da infraestrutura em Nuvem em tempo real, diversas informações são disponibilizadas pelos *gerenciadores de nuvem (GN)*, isto é alvo de monitoramento com grande possibilidade de aplicações que vão desde a mineração das informações para a tomada de decisão em relação aos estados da Nuvem através de uma base de conhecimentos, automação de processos na Nuvem como a simples necessidade de iniciar uma máquina virtual que disponibiliza um novo serviço e evitando a degradação de desempenho e infringir qualquer *ANS*.

Chaves, Uriarte e Westphall (2011 apud ZHANG; CHENG; BOUTABA, 2010) faz referência à analise de alguns dos desafios em termos de pesquisa a ser realizada em torno do paradigma de *CN*. Um deles é o que faz menção à um dos escopos deste trabalho, ou seja, a necessidade de melhores ferramentas para análise e gerenciamento das Nuvens.

A necessidade de monitoramento e as características principais de um sistema de monitoramento para a Nuvem são definidos na *Taxonomia*[3] de Aceto et al. (2013), analisando os atuais arcabouços de monitoramento em nuvem e avaliando suas funcionalidades de acordo com as propriedades básicas a um sistema de *monitoramento em Nuvem (MN)*.

[3] Teoria ou nomenclatura das descrições e classificações científicas.<http://www.priberam.pt/dlpo/taxonomia>

Aceto et al. (2013) elenca várias propriedades que considera essenciais para que uma ferramenta de monitoramento adira, ou seja, atenda aquela “qualidade” e efetivamente seja uma ferramenta robusta e eficaz no monitoramento de Nuvens, sua aplicabilidade é melhor explicada na seção de ferramentas de monitoramento do Capitulo 3. Mas as propriedades são: escalabilidade, elasticidade, resiliência, confiabilidade e avaliabilidade, adaptabilidade, pontualidade, autonomia, extensibilidade, intrusividade e acurácia.

Relacionado a isso Schubert (2014 apud ACETO et al., 2013), escreve o seguinte:

> “[. . .] as plataformas e ferramentas tradicionais para monitoramento, comumente utilizadas para o monitoramento de redes não satisfazem as propriedades de um sistema de monitoramento para nuvens, da mesma forma, as ferramentas e plataformas de monitoramento que se destinam especialmente a Nuvens apesar de apresentar melhor aderência, ainda carecem atender [. . .]. Ademais, o expressivo crescimento no volume e complexidade de dados de monitoramento necessita de tratamento aderente as necessidades dos provedores e consumidores da nuvem, como analise em tempo real de fluxos de dados, mineração e extração de informações relevantes ao contexto e construção de conhecimento. [. . .]”

Portanto, o que se pode concluir a respeito desta afirmações é que há ainda um caminho a se percorrer na adequação das ferramentas de *monitoramento em nuvem (MN)* principalmente no que se refere ao **fluxo de dados das Nuvens**, **mineração dos dados** e **construção de conhecimento** a partir destes dados e que as soluções existentes não são projetadas para lidar com este novo formato de dados de ambientes de Nuvem, neste ponto é onde se enquadra a proposta de uma nova arquitetura de monitoramento em Nuvem, proposta por Schubert (2014), a analise da adequação das ferramentas atuais no contexto de monitoramento de uma Nuvem e não só de Redes, bem como o principal objetivo desse trabalho que é a proposição, exemplificação e de um sistema de banco de dados e um esquema para o mesmo, justificando o porquê dessas escolhas.

1.1 ENQUADRAMENTO

Depois do exposto e dentro deste contexto da dissertação de Schubert (2014), o âmbito deste trabalho é percorrer todos conceitos relacionados à arquitetura cognitivo-autônoma de monitoramento em Nuvem e analisar o ferramental de monitoramento atual, classificando-o de acordo com a taxonomia de monitoramento proposta em Aceto et al. (2013) e analisando sua adequação dentro deste contexto. O objetivo final é analisar, exemplificar e propor um *modelo de dados* temporal-intervalar em séries temporais para

a arquitetura de monitoramento desenvolvida no âmbito do LRG. Preocupando-se com as opções e adaptações possíveis para o *esquema* em Apache HBase™, bem como um possível estudo de caso desta infraestrutura de dados para receber métricas geradas pela Nuvem.

1.2 OBJETIVO GERAL

Propor um *modelo de dados NoSQL* intervalar-temporal em séries temporais para o Apache HBase™ exemplificando sua utilização para as *métricas de monitoramento* fornecidas pelo *gerenciador da nuvem* do *LRG* (Laboratório de Redes e Gerência) [4], Apache CloudStack™ e a plataforma de monitoramento de redes Zenoss™.

1.3 OBJETIVOS ESPECÍFICOS

Os objetivos específicos deste trabalho, são:

- Exemplificar com exemplo o mapeamento de alguma métrica de monitoramento segundo relevância e, possibilidade de utilização no *estudo de caso* da escolha de *chaves linha (row-keys)* para o modelo de dados, de modo a indicar que na migração de um esquema *MySQL*® para o HBase™ nenhum dado será perdido;
- Apresentar os conceitos aderentes à arquitetura de Nuvem proposta em Schubert (2014) como: a computação em nuvem, autonôma (CA), cognitiva (CC) e utilitária (CU);
- Avaliação do ferramental de monitoramento em nuvem(MN) e seu enquadramento no contexto da arquitetura proposta e segundo Aceto et al. (2013);
- Apresentar os conceitos de banco de dados relacionais (BDR) e *NoSQL*;
- Avaliação da utilização de banco de dados (BD) *SQL* versus *NoSQL* na implementação do problema proposto;
- Fundamentar o porquê da escolha do Apache HBase™ para uma solução de banco de dados em séries temporais e intervalar-temporal;
- Definir a chave da linha para realizar as consultas básicas no contexto de monitoramento em Nuvem e justificar o porquê dessa escolha no estudo de caso.

[4] <http://www.lrg.ufsc.br>

1.4 ORGANIZAÇÃO DO TRABALHO

Capítulo 1: Apresenta introdução, enquadramento, assim como: o objetivo geral, objetivos específicos e trabalhos correlatos.

Capítulo 2: Apresenta todo arcabouço teórico relacionado à arquitetura de monitoramento utilizada neste trabalho para justificar um modelo de armazenamento em massa de dados.

Capítulo 3: Apresenta as tecnologias e conceitos relacionados à computação em Nuvem, o conceito de monitoramento em Nuvem e análise de uma taxonomia para efetuá-lo com as ferramentas adequadas, indicando a classificação dessas ferramentas de acordo com as propriedades endereçadas por esta taxonomia.

Capítulo 4: Apresenta o conceito de big data (massa de dados) e todos os bancos de dados envolvidos para a migração de um paradigma relacional para um temporal-intervalar com séries temporais no monitoramento em Nuvem.

Capítulo 5: Trata do *modelo de dados* proposto, escolha da *chave da linha*, e um estudo de caso para a sua escolha, por fim é feito um experimento comparando a performance em transações para ambos banco de dados envolvidos neste trabalho.

Capítulo 6: Apresenta a conclusão, assim como descreve possíveis trabalhos futuros.

1.5 TRABALHOS CORRELATOS

Os trabalhos correlatos apresentados a seguir, que tratam respectivamente do *Monitoramento em Nuvem (MN)* e *big data* , não exploram no todo a abordagem deste projeto, mas partes relevantes do que propõem são complementares neste trabalho de graduação e são considerados como referência. No entanto, como se pode ver pelas datas, este assunto é muito recente e de pesquisa no estado da arte.

- (PALHARES, 2012) - Esta dissertação apresenta uma Abordagem Estratificada ao Monitoramento de Serviços na Nuvem. O objetivo principal é a identificação das várias dimensões deste Monitoramento, combinando as perspetivas do fornecedor de

infraestruturas, de serviços, e dos clientes. Este processo envolve a identificação de Parâmetros e Métricas relevantes para cada dimensão monitorada. Como ambiente de testes utilizou-se a Nuvem implementada no Departamento de Informática da Universidade do Minho e foi utilizado um dos testes em curso, onde estava sendo utilizado o HBase™. O objetivo do mesmo foi avaliar o Apache HBase™ num ambiente *multi-inquilino* característico de um ambiente em nuvem utilizando um gerador de cargas de trabalho. Para a captura de dados estatísticos foram utilizadas ferramentas de monitoramento Dstat© e Ganglia©.

- (HADJIGEORGIOU, 2013) - Esta dissertação têm os dois *sistemas de banco de dados (SGBD)* e compara-os, tentando encontrar um meio termo, onde suas implementações são tão próximas quanto possível e os geradores de carga de trabalho que foram executados não favoreçam um ou outro sistema. O objetivo principal foi encontrar um conjunto de dados e representá-lo em ambos os bancos de dados de forma eficaz. Além disso, a escolha correta das cargas de trabalho foi essencial.

2 CONCEITOS E ARQUITETURA DE MONITORAMENTO

2.1 COMPUTAÇÃO UTILITÁRIA (CU)

O conceito fundamental da *computação utilitária* é que os serviços básicos como *armazenamento*, *processamento* e *largura de banda* de uma rede são tratados como uma mercadoria através de provedores especializados com um baixo custo por unidade utilizada. Usuários destes serviços não precisam se preocupar com *escalabilidade*, pois o que se propõe é fornecer disponibilidade total. Os tempos de resposta não dependem do número de usuários simultâneos, do tamanho do banco de dados ou de qualquer outro parâmetro. Os usuários não precisam se preocupar com *backups*, pois se os componentes falharem, o provedor é responsável por substitui-los e tornar os dados logo disponíveis por meio de réplicas. A principal razão para a criação de tais serviços baseados em *computação utilitária* é o pagamento apenas pelos recursos que foram efetivamente utilizados, portanto, não é necessário um grande investimento inicial por parte do provedor de serviço e o custo cresce de forma previsível. Os prestadores de serviço em seguida, tentam maximizar sua própria utilidade, pois o lucro está diretamente relacionado com o tipo de serviço oferecido, (BUYYA; BROBERG; GOSCINSKI, 2011).

Tradicionalmente, para tratar *picos de carga*, as organizações muitas vezes projetavam os *datacenters* com poder de processamento suficiente para gerência-los, o que significa que para a maioria do tempo os *datacenters* não foram totalmente utilizados. Ao utilizar Nuvem, uma organização pode construir um centro de dados (datacenter) com as especificações que permitirá a entidade executar todas as cargas de trabalho normal do dia a dia dentro de seu ambiente e, em seguida, usar provedores de computação em Nuvem para fornecer recursos adicionais e gerenciar os picos de carga. A *computação utilitária* é freqüentemente associada a algum tipo de plataforma de virtualização que permite que uma quantidade quase infinita de *armazenamento* e *processamento* possa ser disponibilizado para as plataformas de usuários.

A evolução da CN é agora expandir a definição de *computação utilitária* para incluir novos serviços além dos de infraestrutura. O caso para este trabalho seria aproveitar o conceito de *computação utilitária* na auto-gestão da Nuvem utilizando a arquitetura de

monitoramento proposta, e os outros conceitos envolvidos para minerar dados e tomar decisões alinhadas às *regras de negócio* e o ciclo de análise da construção do estado final da Nuvem. Para tal é preciso pensar em utilidade para evitar a violação de *acordo de nível de serviço*, picos de carga e com o fim de monitorar a Nuvem e adaptar as ferramentas utilizadas para monitoramento à realidade da Nuvem que gera uma grande *massa de dados* quer seja a nível de métricas de monitoramento, ou informações provenientes dos clientes nos serviços prestados.

2.2 ACORDO DE NÍVEL DE SERVIÇO (ANS/SLA)

Numa definição mais simples, um ANS/SLA é um contrato entre um fornecedor de serviços de *TI* e um cliente especificando, no geral em termos mensuráveis, quais serviços o fornecedor vai prestar. Níveis de serviço são definidos no início de qualquer relação de *outsourcing* (terceirização) e usados para mensurar e monitorar o desempenho de um fornecedor.

No caso da Nuvem, os ANSs são fornecidos por provedores de IaaS para declarar um compromisso com a entrega de determinada *Qualidade de Serviço (QoS)*[1]. No lado do cliente, isto é uma garantia pois um ANS geralmente inclui a *disponibilidade* e *desempenho*. Além disso, medidas devem ser acordadas por todas as partes, bem como as penalidades pela violação ou não cumprimento dessas expectativas. A maioria dos provedores IaaS concentram seus termos de SLA na garantia de *disponibilidade*, especificando o percentual mínimo de tempo que o sistema estará disponível durante um determinado período.

Um exemplo de SLA é o fornecido pelo serviço a Amazon EC2© em que diz que, se o tempo de funcionamento anual para um cliente cai abaixo de 99,95% para o ano de serviço, o cliente é elegível a receber um crédito de serviço equivalente a 10% do valor acertado pelo serviço, Buyya, Broberg e Goscinski (2011, tradução nossa).

2.3 COMPUTAÇÃO AUTÔNOMA (CA)

Inicialmente proposta como uma possibilidade nos idos de 2001. A *computação autônoma (CA)* surgiu da limitação humana de processar, administrar e *tomar decisões* sobre *sistemas computacionais (SC)* cada vez mais complexos, heterogêneos e integrados com

[1] Capacidade de fornecer um serviço conforme às exigências de tempos de resposta e de banda concorrida.

requisitos estritos de tempo, perfomance e qualidade de informação apresentada. Portanto, trata-se de uma metodologia de construção de *SCs* dotados de mecanismos de *auto-gerenciamento*, dirigido através de informações de auto-nível e objetivos proporcionados por um operador humano, (SCHUBERT, 2014).

Os provedores de CNs devem estar aptos a atender *demandas imprevisíveis*, gerenciando da melhor forma possível a disponibilidade dos seus recursos físicos e virtuais. Além disso, a Nuvem também deve ser capaz de adaptar automaticamente a configuração da sua infraestrutura com o objetivo de evitar violações nos requisitos dos serviços SLAs de seus usuários.

Em Kephart e Chess (2003), os autores definem que a *computação autonômica (CA)* é um ambiente computacional com habilidade de se gerenciar e dinamicamente se adaptar a mudanças de acordo com políticas e objetivos pré-estabelecidos. Os ambientes auto-gerenciáveis são capazes de, autonomicamente, executar suas atividades baseados no estado observado, sem requerer alguma entidade controladora. Essas tarefas são realizadas em um ciclo de controle, por vezes denominado *MAPE-K (Monitor, Analyse, Plan, Execute, and Knowledge)*, o diagrama na Figura 1 a seguir ilustra o modelo.

Figura 1 – Ciclo Autônomo MAPE-K, (SCHUBERT, 2014).

Devido à estas características, a *CA* apresenta um projeto de entidades e características interessantes aplicáveis ao ambiente da Nuvem. Considerando estes aspectos,

nuvem autônoma (NA) é definida na dissertação de Schubert (2014), como uma Nuvem com autonomia para *tomar decisões* importantes sobre *gerenciamento* e *alocação de recursos* apropriados para atender as requisições dos usuários e as *decisões de otimização* da utilização dos recursos.

Em Schubert (2014 apud BUYYA; CALHEIROS; LI, 2012) a *Computação em Nuvem Autônoma (CNA)* é apresentada como um modelo na obtenção de infraestruturas em Nuvem confiáveis, seguras e com um custo atrativo. Além disso menciona alguns pontos importantes, que fundamentam o monitoramento eficiente da Nuvem, proporcionando uma base sólida para análise e *tomada de decisões* a partir de *elementos autônomos*.

Qualidade de Serviço: Os provedores em Nuvem devem garantir a quantidade e qualidade de recursos necessária a ser provisionada para os clientes mantendo as limitações de orçamento e custo.

Eficiência Energética: Uso eficiente da energia na infraestrutura evitando a utilização desnecessária de recursos para aplicações e minimizando o impacto de CO2.

Segurança: Atingir padrões de confidencialidade, disponibilidade e confiabilidade contra *ataques de negação de serviço(DoS Attack)*[2].

De acordo com (ENDO; SADOK; KELNER, 2011), as *nuvens autônomas (NA)* emergem como o resultado da aplicação das propriedades de auto-gerenciamento advindas da *CA*: *auto-configuração*, *auto-cura*, *auto-otimização* e *auto-proteção*.

Técnicas, como computação bio-inspirada, sistemas multi-agente, técnicas evolucionárias e várias outras são bastante focadas, pois, além de se adequarem muito bem a cenários distribuídos, podem ser utilizadas com o objetivo de tornar o sistema mais robusto, adaptável e melhorar aspectos da Nuvem, como gerenciamento e otimização de recursos, e manutenção e minimização de custos operacionais. Portanto, as *NA* surgem da conveniência tecnológica em se utilizar as propriedades dos ambientes auto-gerenciáveis da *CA* em CN, (SCHUBERT, 2014).

[2] É uma tentativa em tornar os recursos de um sistema indisponíveis para seus utilizadores. Não se trata de uma invasão do sistema, mas sim da sua invalidação por sobrecarga.

2.4 COMPUTAÇÃO COGNITIVA (CC)

O conceito por trás da *computação cognitiva (CC)* é essencialmente um sistema que é capaz de aprender através do uso. Entretanto, o que torna a idéia particularmente poderosa é a combinação desta capacidade de aprendizagem com grandes conjuntos de dados, big data. Porque o cérebro humano não tem a mesma capacidade de armazenar e recuperar grandes quantidades de informação com precisão. Mas, a questão principal consiste em como para recuperar estes dados de uma forma que é útil no mundo real, para resolver os problemas humanos. Portanto, o objetivo final é um processo de humanização da computação onde a informação armazenada é alvo de um processamento inteligente que sugere ações para determinada questão ou ser independentemente atuante em situações em que for necessário. O resultado é o auxílio por parte do sistema em um processo de *tomada de decisão (TD)*.

Segundo Schubert (2014), apesar do grande conhecimento a respeito do domínio em causa, por ir à fundo a respeito das informações massivas, tal sistema não tem o propósito de susbtituir especialistas humanos, mas sim atuar como um *Sistema de Suporte a Decisão (SSD)*[3], além disso:

> Pode ser afirmado que a *computação cognitiva (CC)* e uma abordagem contemporânea à preocupação de tratarmos a incerteza e utilidade agregando novas técnicas como aprendizagem de máquina e processamento de linguagem natural, endereçando conceitos da *teoria da decisão(TD)*, (SCHUBERT, 2014)

2.5 ARQUITETURA COGNITIVO-AUTÔNOMA DE MONITORAMENTO EM NUVEM (ACAMN)

A CN e toda a abstração complexa intrínseca ao seu modelo será ignorada pelo usuário final devido a implementação da autonomia e cognição em sua arquitetura, com efeito não apenas no *MN*, como também, em seus recursos. Alinhado aos fundamentos vistos nas seções anteriores de CC e CA, é possível inferir de Kephart e Chess (2003), que o próximo grande avanço a ser obtido é automação e construção de sistemas heterogêneos e auto-gerenciados e, mais recentemente uma extensão da auto-gerência com a *CC*. Dentro desse contexto Schubert (2014) propôs uma *arquitetura cognitivo-autônoma de monitoramento em Nuvem (ACAMN)* onde há:

> . . . crescente automação e construção de sistemas inteligentes, que constuitem-se em uma exigência para sistemas distribuidos de larga escala com restrições

[3] Refere-se simplesmente a um modelo genérico de tomada de decisão que analisa um grande número de variáveis para que seja possível o posicionamento a uma determinada questão

> de tempo e necessidade de decisão e reação dinâmicas e instantâneas para manter um estado saudável, (DOBSON et al., 2010, tradução nossa).

À estes conceitos já indicados, adicionando a taxonomia de monitoramento(TM), principalmente a propriedade de *Pontualidade* e as possíveis trabalhos em nível de *monitoramento de infraestruturas (MI)* é possível prever possíveis violações de SLA. A proposta de uma arquitetura autônoma e cognitiva de monitoramento visa remover a maior parte da intervenção humana do proceso de gerenciamento da Nuvem, e prover informação mais acurada sobre o seu estado, (SCHUBERT, 2014), evidenciada na Figura 2:

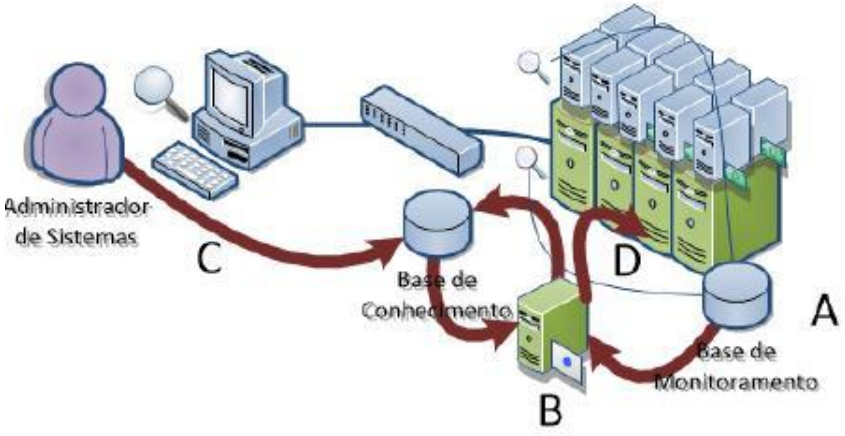

Figura 2 – Arquitetura de Monitoramento em Nuvem PRIVADA proposta por, (SCHUBERT, 2014).

A) *Base de Monitoramento:* Aderindo aos princípios de CA, os elementos do modelo MAPE-K mostrados neste capítulo devem ser auto-gerenciáveis. A principal diferença do modelo convencional é a centralização das *BC*, ao contrário do que se passa quando há várias *FM*.

B) *Etapas MAPE-K:* Consiste basicamente na *Análise*, *Planejamento* e *Execução*. Onde o primeiro é analise dos dados contidos em (A), clusterização e subsequente classificação dos mesmos. Análise das regras de negócio, validação do ANS e finalmente a construção do estado final da Nuvem. O segundo consiste na aplicação das regras de (A) caso haja violações de ANS. O terceiro e último é a aplicação das ações no sistema.

C) *Intervenção Humana:* É alimentação das *Regras de Negócio(RN)* e *Base de Conhecimento(BC)* com regras de Monitoramento, ANS/SLA, objetivos da Nuvem e critérios de alocação e desalocação de recursos.

D) *Execução:* Execução de ações sobre a Nuvem e, posterior registro de ações na *Base de Conhecimento(BC)*.

3 COMPUTAÇÃO EM NUVEM, MONITORAMENTO E FERRAMENTAS

3.1 VIRTUALIZAÇÃO

Uma das principais tecnologias que possibilitou o surgimento do paradigma de CN é a *virtualização*, que tem a capacidade de fornecer uma abstração dos recursos de computação. Hoje em dia as plataformas de infraestrutura são predominantemente de dois tipos, que são os ambientes *totalmente virtualizados* ou *para-virtualizados* (KING; FORD, 2013, grifo nosso).

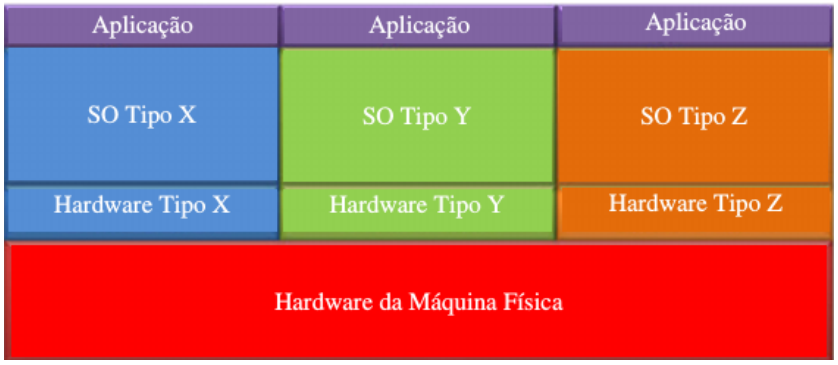

Figura 3 – Ambiente de Virtualização Emulado, (CHANTRY, 2009).

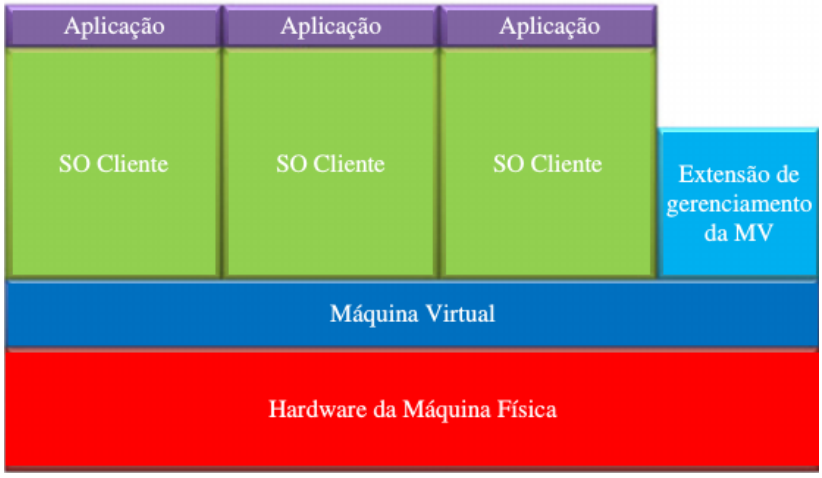

Figura 4 – Ambiente de Para-virtualização, (CHANTRY, 2009).

Não é uma tecnologia recente, já é utilizada nos *mainframes* desde a década de 1970. A *virtualização* de servidores proporciona redução nos custos de aquisição dos servidores físicos, facilita a administração, diminui os recursos de infraestrutura necessários para hospedar os servidores e o consumo de energia. Há outras variações para virtualização, porém os três tipos representados pelas figuras 3, 4 e 5, são os tipos mais comuns e usados

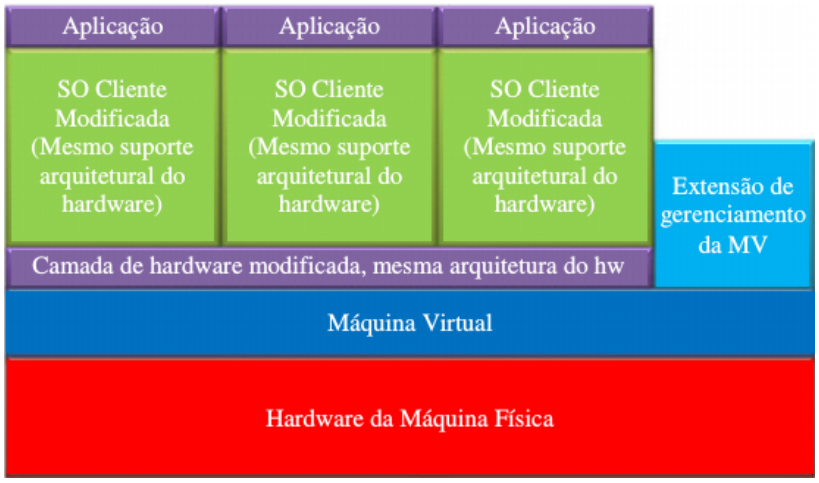

Figura 5 – Ambiente de Virtualização Total, (CHANTRY, 2009).

na CN. Um hipervisor, como veremos na próxima Seção, não utiliza necessariamente um único modelo, o mais importante é saber que estes conceitos existem e o que eles significam.

Emulação – O ambiente virtual emula uma arquitetura de *hardware (HW)* exigida de outro *sistema operacional (SO)*. Um dos casos comuns em que você encontra o *HW* emulado é com dispositivos móveis. Os desenvolvedores de aplicativos usam um ambiente emulado para testar e desenvolver os aplicativos para dispositivos móveis em geral.

Para-virtualização (PV) - O hipervisor exporta uma cópia modificada do *HW*. A *camada de exportação* tem a mesma arquitetura que o *HW* do servidor. A consequência disso reflete no desempenho porque é utilizado os *drivers* reais.

Virtualização total (VT) - Uma imagem completa de outro SO é feita e executada dentro de um ambiente virtualizado. A diferença entre a *virtualização total (VT)* e *emulação* é que todos os convidados virtuais executam na mesma arquitetura de *HW*. Todos os clientes suportam o mesmo *HW*, o que permite que o cliente execute várias instruções diretamente no *HW*, consequentemente irá proporcionar um melhor desempenho.

3.1.1 Hipervisor

Segundo Chisnall (2008), um hipervisor, também chamada de *virtual machine manager (VMM)*, é uma das muitas técnicas de virtualização de *HW* que permitem que vários *SOs*, denominados convidados, possam executar simultaneamente em um computador *host*. Este monitor de *máquina virtual (MV)* fornece uma interface baseada em estreita colaboração com a arquitetura física subjacente. É assim chamado porque é conceitualmente um nível mais elevado do que um programa de supervisão. O hipervisor apresenta para

os *SOs* convidados uma plataforma operacional virtual e gerencia a execução dos SOs convidados. Várias instâncias de uma variedade de SOs podem compartilhar os recursos de *HW* virtualizados.

Hipervisores estão instalados no *HW* de servidor cuja única tarefa é executar *SOs*. *sistemas de virtualização (SV)* não hipervisor são utilizados para tarefas semelhantes em dispositivos de servidor dedicado, mas também comumente no ambiente de trabalho, computadores portáteis e até *desktops*. O termo é frequentemente usado para descrever a interface fornecida pela infraestrutura de CN, neste caso, na funcionalidade específica como um serviço informação como serviço (IaaS).

Mais precisamente, na virtualização completa da *VMM* existe uma interface idêntica à arquitetura física subjacente. Isto tem a vantagem de que os sistemas de exploração atuais possam executar de forma transparente e não modificada no monitor de *MV*.

Segundo Chisnall (2008), o que a experiência tem mostrado, no entanto, que a virtualização completa pode ser difícil de desempenhar com desempenho satisfatório em muitas arquiteturas de computadores, incluindo a família de processadores x86, e que o desempenho pode ser melhorado, permitindo uma interface modificada sendo fornecida (com a desvantagem de que *SOs* precisam então de ser portados para esta interface modificada), o que é conhecido por *Para virtualização (PV)*.

O exemplo mais utilizado para a implementação de *VMMs* é o XenServer™, que, a partir da sua versão 3.0, passou a oferecer a *Virtualização Total (VT)*, desde que seja executado sobre um *HW* com suporte a virtualização nativamente.

3.1.1.1 XenServer®

O hipervisor XenServer® é um projeto de pesquisa da *Universidade de Cambridge*, liderado pelos professores *Ian Pratt* conjuntamente com *Simon Crosby*, sendo seu primeiro lançamento público em 2003, (XENSERVER, 2014, tradução nossa).

O XenServer® é fundamentalmente um pacote que consiste em serviços de: hipervisor, *toolstack (pilha de ferramentas)*, domínio e controle de apoio que se combinam para produzir uma plataforma de virtualização. Hoje existem duas versões distintas do XenServer®, uma comercial conhecido como XenServer6.2® e outra de código totalmente aberto (originalmente conhecido como XCP), que na verdade começou há alguns anos atrás

mas foi liberada novamente em 24 de Junho de 2013 (XENSERVER, 2014, tradução nossa).

Em resumo, é um *"middleware" (MW) (mediador)*[1] que roda sobre a camada de *HW* substituindo o *SO* e permitindo que vários outros SOs virtualizados rodem corretamente. O *gerenciador da nuvem (GN)* do LRG, o CloudStack™ pode utilizar diferentes tipos de hipervisores, mas na sua implementação atual utiliza o XenServer®.

Como descreve Sabharwal e Shankar (2013), dependendo do tipo de hipervisor, a forma como é implementado varia, neste caso é criado um *pool master (pool principal)*[2] utilizando o cliente XenCenter© que contém vários *hosts* (hospedeiros). Enquanto da criação do *pool* é adicionado todos *slave hosts (hospedeiros escravos)* no XenServer® dele. O sistema de controle das *MVs* e a administração da sua rede virtual funcionam todos em nível de *hosts*, como se pode ver na Figura 6 abaixo.

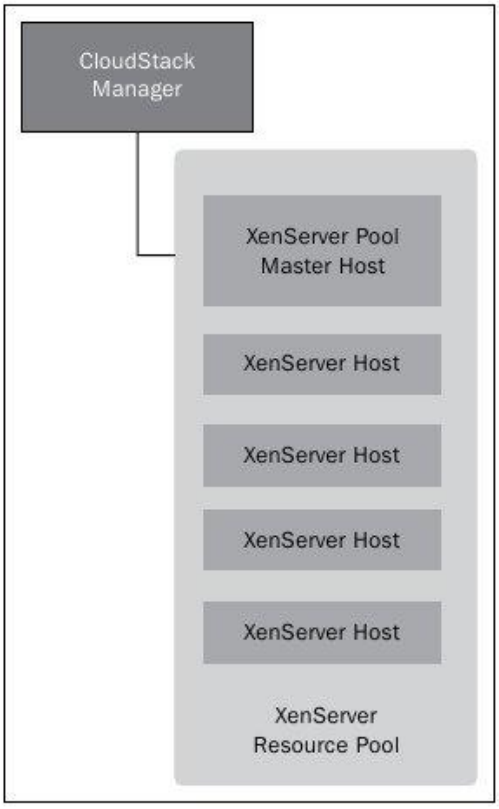

Figura 6 – Implementação do XenServer™ no CloudStack™, (SABHARWAL; SHANKAR, 2013).

[1] No campo da computação distribuída, é um programa de computador que faz a mediação entre software e demais aplicações. É utilizado para mover ou transportar informações e dados entre programas de diferentes protocolos de comunicação, plataformas e dependências do sistema operacional.

[2] Conjunto de recursos que são mantidos inicializados prontos para uso e, são atribuídos e destruídas sob demanda.

3.2 CLUSTERS

É definido como um sistema que compreende dois (2) ou mais computadores ou sistemas, denominados nodos ou nós, que trabalham em conjunto para executar aplicações ou realizar outras tarefas de maneira transparente aos usuários. Ou seja, os usuários terão a impressão de estar utilizando um único sistema.

Basicamente, existem quatro tipos de clusters amplamente conhecidos e utilizados: *Alta Disponibilidade*, *Balanceamento de Carga*, *Combinação Alta Disponibilidade* e *Balanceamento de Carga* e *Processamento Distribuído* ou *Processamento Paralelo*, (PITANGA, 2004).

Na CN, utiliza-se clusters de *Alta Disponibilidade* a fim de garantir a disponibilidade, a escalabilidade e a elasticidade dos recursos. Cada nó do cluster de alta disponibilidade será um hospedeiro de MVs e garantirá a disponibilidade dos recursos através de *failover*, isto é, quando um nó do cluster falhar, o serviço estará disponível em outro nó.

3.3 COMPUTAÇÃO EM NUVEM (CN)

Apesar de ser considerado um paradigma atual de tecnologia, a *Computação em Nuvem (CN)* foi citada acadêmicamente pela primeira vez em 1997 pelo professor de *Sistemas de Informação (SI)*, *Ramnath Chellappa*. Entretanto, já se ouvia falar do assunto na década de 1960 a partir de pioneiros como *J.C.R Licklider*, uma das grandes influências do *Advanced Research Projects Agency Network (ARPANET)*[3], e *John MacCarthy* que definia a computação como uma *utilidade pública*. Portanto, o conceito fundamenta em si surgiu muitas décadas atrás.

A primeira utilização de fato foi em *transações financeiras* e tratamento de dados de *censos*, mas dentro de poucos anos as empresas trocaram os dispositivos físicos por serviços em Nuvem devido à grande vantagem de redução de custos em material e pessoal, assim como, simplificação da obtenção dos serviços necessários o que resulta em eficiência para os desenvolvimentos dos trabalhos em uma empresa.

Em uma publicação da Dell®, na Forbes®, Cantu (2011) escreve o que vêm a seguir:

[3] Projeto do Departamento de Defesa dos Estados Unidos, foi a primeira rede operacional de computadores à base de comutação de pacotes, e o precursor da Internet.

Figura 7 – Modelo de Computação em Nuvem, (BUYYA; BROBERG; GOSCINSKI, 2011).

> De acordo com as projeções da Century Link®, até 2015, o mundo testemunhará um aumento de quatro vezes na quantidade de dados criados e replicados. E assim que surgirem todos esses dados, será necessária uma forma para armazená-los com segurança e permitir que os usuários finais tenham acesso a eles de forma eficiente.

Outro grande propulsor desta indústria de serviços é o crescimento explosivo de dados, o que está levando esta tecnologia às nuvens liretalmente é o *big data*. Juntamente com o *monitoramento em nuvem (MN)*. *big data* e *computação em nuvem (CN)* são as tecnologias que ainda não atingiram o grau de maturidade desejado e estão quase sempre relacionadas às dificuldades de processamento encontradas ao utilizar infraestruturas convencionais para o processamento de grandes volumes de dados.

3.3.1 Definição

Autores diversos e de referência definem o paradigma da CN com propostas diferentes, o seu modelo mais aceito é como se pode ver na Figura 7. A definição de *CN* de acordo com Mell e Grance (2011, tradução nossa), do National Institute of Standards and Technology (NIST)[4], é a seguinte:

> Computação em Nuvem é um modelo que possibilita acesso, de modo conveniente e sob demanda, à um conjunto de recursos computacionais conjuráveis (por exemplo, redes, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente adquiridos e liberados com mínimo esforço gerencial ou interação com o provedor de serviços.

[4] É uma agência governamental cuja missão é promover a inovação e a competitividade industrial dos Estados Unidos, promovendo a metrologia, os padrões e a tecnologia de forma que ampliem a segurança econômica e melhorem a qualidade de vida.

Outra definição também interessante, porém a título de reforço do conceito, é a de Buyya, Broberg e Goscinski (2011, tradução nossa):

> A Nuvem é um tipo de sistema distribuído e paralelo composto de uma coleção de computadores interconectados e virtualizados que são dinamicamente provisionados e apresentados como um ou mais recursos computacionais unificados baseado em acordos de níveis de serviço ANSs estabelecidos através de provedores de serviços e consumidores.

Embora a definição de CN possa parecer complexa, num primeiro instante, ela foi projetada para que seus usuários pudessem utilizá-la da maneira mais simples quanto possível. A infraestrutura projetada para suportar o Ambiente em Nuvem normalmente é constituída de diversas máquinas físicas de baixo custo que estão dispostas em uma rede, e podem ser divididas entre *máquinas virtuais (MV)* (processamento) e armazenamento, formando os conhecidos *datacenters*, como mostrado na Figura 8.

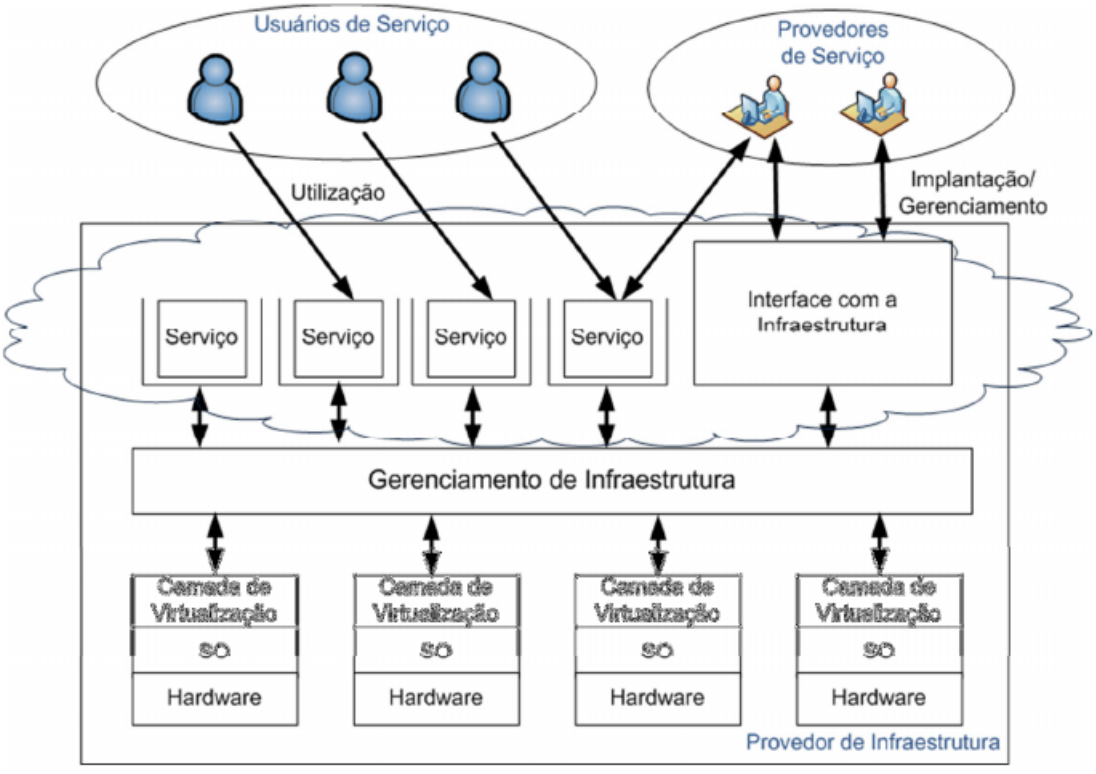

Figura 8 – Arquitetura de uma Nuvem, (BUYYA et al., 2009).

3.3.2 Características

As principais características da CN são de fornecimento de serviços *armazenamento* e *processamento de dados*, onde os clientes utilizam aplicativos que consomem

bastante processamento e memória, o que é bastante versátil pois não utiliza recursos da máquina do usuário para tais tarefas. Estas são classificadas da seguinte maneira:

Self-service : Aquisição de recurso computacional de forma unilateral por parte do usuário, com o objetivo de atender necessidades momentâneas ou permamentes sem a necessidade de intervenção humana para contactar os provedores de serviços. Portanto, é possível fornecer um serviço de qualidade e automático de gestão de recursos computacionais que disponibilizará toda sua capacidade de armazenamento, tempo de processamento ou outros recursos sob demanda.

Amplo Acesso :Em termos gerais é o compartilhamento de recursos através de uma Rede por intermédio de máquinas clientes chamadas *thin clients*. Que são máquinas com servidores de aplicativos destinados a realizar as tarefas mais relevantes de lógica interna e possuem *HW* e *SW* mínimos em sua arquitetura, por isso chamados de *thin*(finos) como por exemplo dipositivos móveis no geral.

Pooling : Os recursos computacionais são disponibilizados em um *pool* de modo que sejam utilizados por diversos usuários, ou seja, uma arquitetura *multi-tenant* (multi-inquilino), possibilitando o acesso a variados recursos físicos e virtuais atribuídos e ajustados conforme a demanda solicitada por cada *tenant* (inquilino) e devida configuração do serviço. A principal característica deste tipo de serviço é o usuário não precisar saber onde fisico-virtualmente se situam os recursos disponibilizados, quer sejam em *datacenters* ou sua localização por país, etc. Com isso, desresponsabiliza-se o usuário de conhecer detalhes de implementação da Nuvem ou serviço, criando uma interface que contém apenas o mínimo em específico para aquele usuário.

Elasticidade : É a característica de fornecer serviços que dão a impressão ao usuário que são *ilimitados*. Pois os recursos são fornecidos de maneira elástica e célere e, em muitos casos de forma automática. Para tal, o provedor deve gerir bem a necessidade de aumento da sua demanda, bem como a retração através da atualização da informação no que diz respeito ao

dinamismo das atividades dos seus clientes. Assim, deve ser capaz de adaptar-se a oscilação ou picos de demanda por recurso.

Medição : Controle automático, monitorado e otimizado da utilização de recursos por meio da capacidade de medição permitindo transparência tanto para o lado do provedor como do usuário. Ou seja, é a automação realizada em algum nível de abstração adequado para determinado tipo de serviço, estes podem ser de: armazenamento, contas de usuário ativas, largura de banda e processamento.

3.3.3 Modelos de Serviços

Os modelos de serviços da CN estão divididos consoante a sua natureza. Nada mais do que representam de forma modular os meios como a CN é oferecida, principalmente no que toca à produtos e serviços ou nichos de mercados diferentes. Os principais são IaaS, PaaS e SaaS, conforme podemos ver na Figura 9. Entretanto, recentemente os padrões estabelecidos pelo *Focus Group on Cloud Computing do International Telecommunications Union - Telecommunication Standardization Sector (ITU-T)*[5] definirem ainda mais dois modelos de serviço, nomeadamente *CaaS (Communications as a Service)* e *NaaS (Network as a Service)*. Eles estão interelacionados de forma que cada um é uma abstração em relação ao anterior, o IaaS é situado na parte mais inferior seguido pelas outras duas plataformas de desenvolvimento. As aplicações são disponibilizadas para o utilizador final e ficam no topo da Nuvem.

As arquiteturas *platform-as-a-service (PaaS)* têm surgido nos últimos anos para aliviar os encargos da gestão de recursos. Este fator aumentou a produtividade para desenvolvedores e suas respectivas organizações. Os desenvolvedores não precisam mais se preocupar com detalhes de nível inferior, como o *consumo de CPU*, as *limitações de largura de banda*, o *consumo de memória* e *uso de disco*, como era comum no passado. A escala de aplicações é agora o ônus do sistema da plataforma. Sistemas *PaaS* se tornaram os sistemas operacionais do *datacenter*.

As arquiteturas de Nuvem, *infrastructure-as-a-service (IaaS)* e PaaS, foram projetadas e otimizadas para serviços web, ou também conhecido como *software-as-a-service*

[5] Um dos três setores da União Internacional de Telecomunicações (UIT) que coordena os padrões para as telecomunicações.

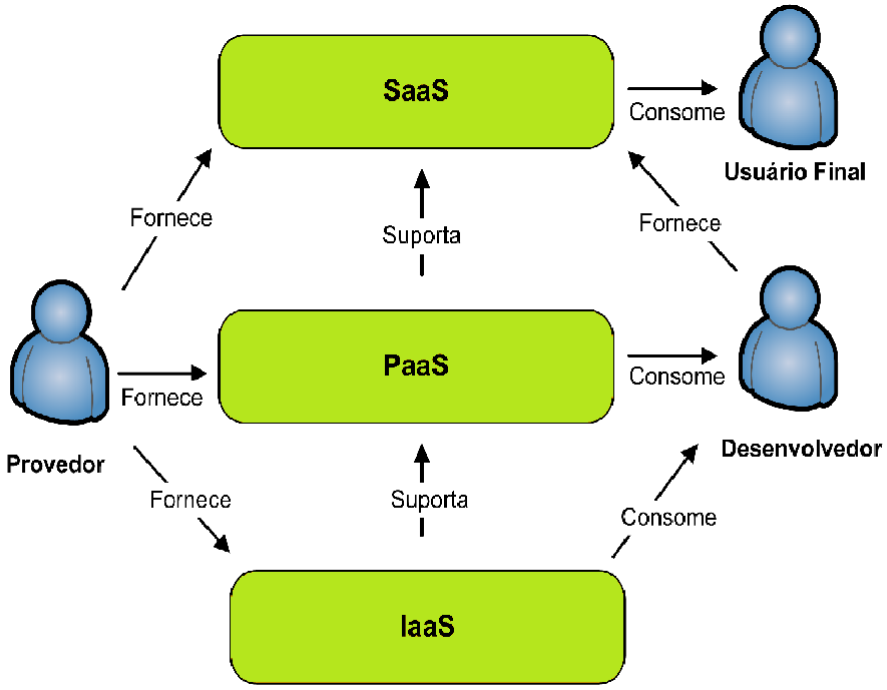

Figura 9 – Modelo de Serviços da Computação em Nuvem, (BRISCOE; MARINOS, 2009).

(SaaS), o domínio de aplicação mais comum. Este paradigma muda o aplicativo de hospedagem do cliente para um serviço remoto, que é acessível através de um navegador web ou através de uma interface *Representational State Transfer (REST)*[6]. Como a popularidade do *(REST)* e *Service Oriented Architecture (SOA)*[7] tem aumentado, o suporte por linguagens de alto nível povoou muitas ofertas para *frameworks (FW)*[8] web como Ruby on Rails™ (RoR), Django para Python™, e Node.js para JavaScript™.

3.3.4 Modelos de Implementação

Para além da classificação anterior, é possível fazer o mesmo com relação a *implementação*, ou seja, o que pode ser considerada a abrangência de público.

Nuvem comunitária (NC) : É uma infraestrutura compartilhada por diversas organizações, dando suporte à uma comunidade com preocupações ou atividades em comum, podendo ser gerenciada pela própria organização ou por terceiros e se localizar dentro ou fora dos limites da organização.

6 Técnica de engenharia de software para sistemas hipermídia distribuídos como a World Wide Web.
7 Arquitetura de software cujo princípio fundamental é que as funcionalidades implementadas pelas aplicações devem ser disponibilizadas na forma de serviços.
8 Abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica. Ao contrário das bibliotecas, é quem dita o fluxo de controle da aplicação, chamado de Inversão de Controle.

Nuvem privada (NPi) : É a infraestrutura operada apenas por uma organização, podendo ser gerenciada pela própria organização ou eventualmente por terceiros e pode tanto existir dentro como fora dos limites da organização. Segundo Chaves, Uriarte e Westphall (2011 apud SOTOMAYOR et al., 2009, grifo nosso):

> ... o objetivo principal de uma *nuvem privada(NPi)* não é vender capacidade pela Internet, através de interfaces acessíveis ao público em geral, mas sim dar aos usuários locais uma infraestrutura ágil e flexível para atender cargas de trabalho de serviços dentro de seu próprio domínio administrativo.

Nuvem pública (NPu) : Disponibilizada para o público em geral ou para um grande grupo industrial no modo *pay-as-you-go*[9].

Nuvem híbrida (NH) : Mell e Grance (2011) definem como uma composição de uma ou mais Nuvens (privada, comunitária ou pública), cada uma das Nuvens permanecem isoladas, as quais são conectadas entre si por tecnologias proprietárias ou padronizadas que permitem portabilidade de dados e aplicações.

3.4 MONITORAMENTO EM NUVEM (MN)

Nos terminais usados na década de 80, a simples falha de um computador central podia paralisar uma empresa inteira, no cenário em Nuvem o pesadelo é o mesmo. Atualmente voltou-se aos antigos conceitos de *TI centralizados*, pois a Nuvem moderna sob este ponto de vista de centralização corresponde ao *mainframe* antigo.

Na mudança do *monitoramento em rede (MR)* tradicional para a *Nuvem*, os responsáveis pela infraestrutura de *TI* devem constatar quais as *demandas de capacidade* das aplicações individuais e quais *flutuações cíclicas esperadas*. Os recursos da Nuvem podem ser planejados com base na avaliação de uma rede de monitoramento global, análises de longo prazo, tendências e picos de carga, são requisitos básicos para assegurar um desempenho constante de *TI* na interação entre diferentes sistemas virtualizados.

Uma Nuvem deve cumprir requisitos em termos de velocidade de transmissão e estabilidade, pois, do contrário, o *HW* ou as conexões de rede deverão ser atualizados. Até mesmo pequenas perdas no resultado da velocidade de transmissão podem levar a grandes

[9] Modelo de prestação de serviços onde se paga apenas por aquilo que é de fato utilizado.

quedas no desempenho geral. Assim sendo, o monitoramento coloca o desempenho da rede no centro das atenções. A queda de uma única *MV* em um ambiente de Nuvem altamente virtualizado pode rapidamente interromper o acesso a várias aplicações centrais. Para garantir que os usuários sempre tenham acesso a aplicativos de negócios terceirizados, é necessário monitorar o desempenho da conexão com a Nuvem em todos os níveis e de todas as perspectivas. Uma solução de *MR* adequada faz tudo isso com um sistema central e alerta imediatamente o administrador de *TI*, tanto no caso de mau funcionamento dentro de sua infraestrutura de *TI*. Isso inclui o monitoramento tanto de cada *MV* como do hipervisor e de todos os servidores físicos, firewalls, conexões de rede, etc.

3.4.1 Conceitos

A definição de uma *taxonomia de monitoramento (TM)* é primordial na CN. Entretanto, a comunidade científica não tem dado a devida atenção, assim como, *provedores* (cuja definição é elemento chave no controle e gerenciamento de recursos computacionais) ou *consumidores* (cuja definição fornece parâmetros de performance para aplicações) de serviços em Nuvem, (SCHUBERT, 2014).

As plataforma de monitoramento da Nuvem na visão do usuário deve ser: escalável, elástica em termos de recursos, prover serviços mensurados e bilhetagem consoante o consumo. Enquanto que na visão dos provedores de serviços deve ser: confiável, com custos de energia reduzidos, apta a cumprir os requisitos de SLAs, segura, (KUTARE et al., 2010). Para tal, a plataforma deve ser alinhada com as características essenciais da Nuvem que está monitorando.

Segundo (BUYYA; CALHEIROS; LI, 2012), elementos que podem somar e serem adaptados a realidade de uma plataforma de monitoramento, provendo auto-monitoramento, e aproximando a CN do desenvolvimento de *sistemas autônomoas(SA)*, são definidos pela CA na seção precedente.

Sob o ponto de vista do monitoramento e comparação com o caso da *computação em Grade (CG)*, o *MN* é mais complexa por causa das diferenças, tanto no modelo de confiança e a visão sobre recursos/serviços apresentados para o usuário,(FOSTER et al., 2009). Na verdade, o principal objetivo de uma *Grade* é a partilha de recursos através de várias organizações Foster e Kesselman (2003), o que implica critérios numéricos mais

simples e de abstração recurso limitado, o que cria uma relação simples entre os parâmetros de monitoramento e status de recurso físico.

Por outro lado, para a *Nuvem*, a presença de múltiplas camadas e paradigmas de serviços conduz a alta abstração de recursos, resultando em relação menos translúcida ou observável entre a camada de serviço e os recursos subjacentes. Esta diferença de objetivos e transparência tem de ser preenchido no momento da adoção na Nuvem de um sistema de Monitoramento que vem da área de CG. Além disso o fato de a *Nuvem* fornecer serviços sob-demanda coloca desafios adicionais para sistemas não projetados para alta agitação dos usuários e recursos de monitoramento.

Ainda segundo Aceto et al. (2013) A arquitetura relativamente rígida da *Grade*, as possibilidades limitadas de negociação de serviços e a baixa automação do provisionamento de recursos fazem os *clusters* comparáveis à uma tecnologia de base para provedores de Nuvem IaaS e levam a exigências em termos de monitoramento que são um subconjunto limitado de os de uma nuvem. Portanto, a maioria das propriedades que caracterizam de sistemas de *MN* ou, não se aplicam para *clusters* ou Grades (ou seja, a elasticidade, adaptabilidade, autonomicidade) ou, não são vitais para a sua finalidade (abrangência, extensibilidade e Intromissão).

3.4.2 Taxonomia de Monitoramento (TM)

Dentro do contexto de arquitetura de monitoramento (AM) apresentada um *Agente* ou *Prova* é um *coletor de dados* sem processamento ou transformação de dado intrínseco, em um ponto ou camada específico da Nuvem, que tem por função a coleta de informações úteis ao *MN* como pode ser vista na Figura 10 e como será referido nas camadas da Taxonomia proposta a seguir.

No modelo proposto pela *Cloud Security Alliance*[10] e revisto por Aceto et al. (2013), a Nuvem divide-se nas seguintes camadas: Facilidades, Rede, *HW*, *SO*, *MW*, Aplicação e Usuário. Estas camadas são os alvos para os *agentes de monitoramento (AM)*[11] responsáveis pelo auto-monitoramento já citado.

Facilidades : Abrange a parte do abastecimento de energia, refrigeração,

[10] Organização sem fins lucrativos com a missão de promover a utilização das melhores práticas para a prestação de garantia de segurança dentro de Cloud Computing.

[11] As três atividades principais envolvidas na administração do agente são descoberta dos dispositivos de destino, implantação ou instalação de agentes nesses dispositivos e gerenciamento contínuo de agentes.

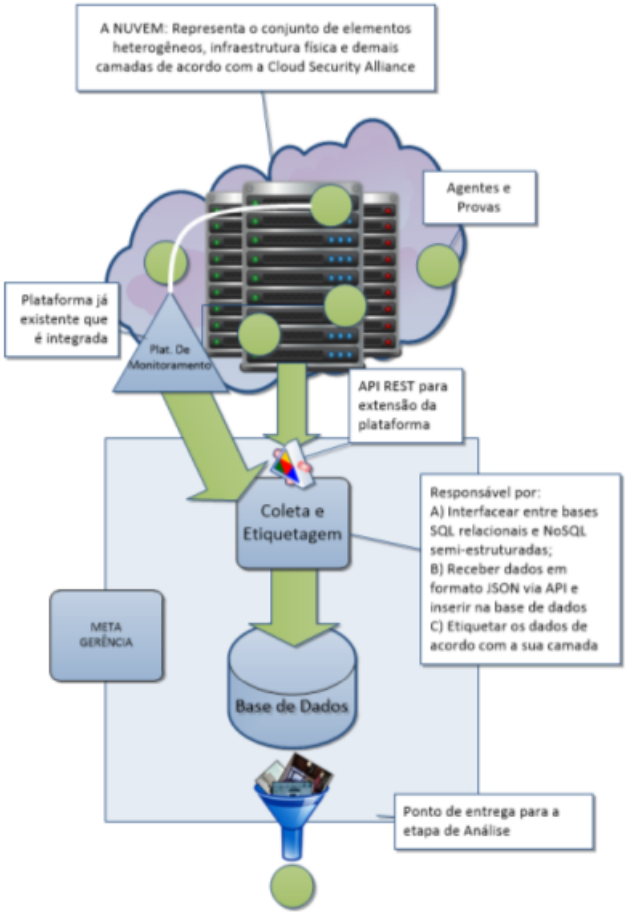

Figura 10 – Arquitetura de Monitoramento em Nuvem proposta por Schubert (2014).

segurança física, centro de dados e demais elementos que compõem a parte física do provedor.

Rede : Caminhos de Dados na Nuvem e entre esta e usuários.

Hardware(HW) : Equipamentos físicos de processamento na Nuvem.

Sistema Operacional (SO) : Sistemas hospedeiros e das MVs.

Middleware(MW) : Solução situada entre o SO e aplicação do usuário.

Aplicação : *SW* executado pelo usuário da Nuvem, normalmente web, e *REST*.

Usuário : Utilizador que acessa a Nuvem.

Segundo a dissertação de Schubert (2014 apud ACETO et al., 2013) existem lacunas nas ferramentas de monitoramento (FM) mostradas a seguir, verificando-se uma baixa adaptação destas para a nova demanda que surgiu com a *CN*, principalmente no que

se refere à *volume de dados* de monitoramento e outras caracterısticas, como *Tomada de Decisão (TD)*.

Um monitoramento distribuído, com *agentes* ou *provas* em várias camadas/métricas é o mais adequado em um sistema complexo e heterogêneo como o da Nuvem e deve ser aderente à um conjunto de propriedades descritas por Schubert (2014 apud ACETO et al., 2013) e que podem ser visualizadas no Diagrama 11, são elas:

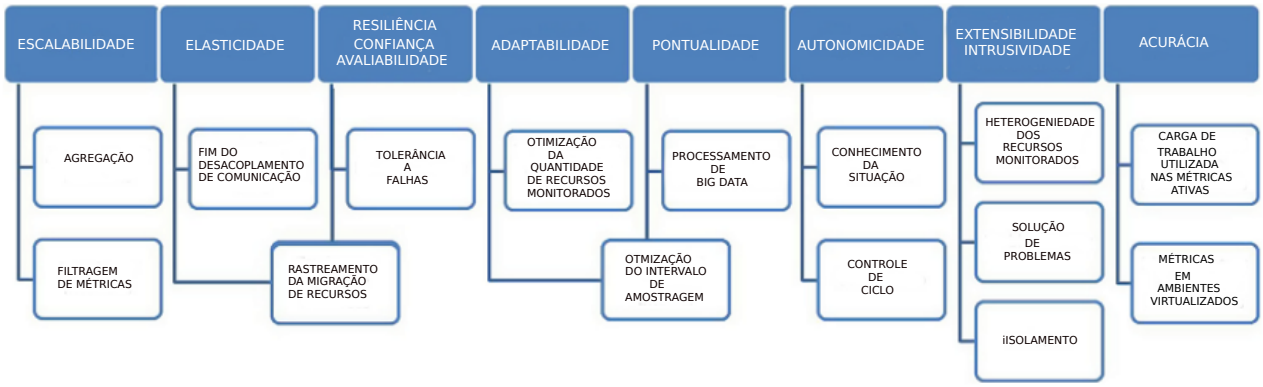

Figura 11 – Taxonomia do monitoramento em Nuvem, (ACETO et al., 2013)

Escalabilidade : Comportar um grande número de dados devido a grande quantidade de parâmetros, conjunto de recursos a ser monitorado cujos sistemas atuais não abrangem, pois limitam o volume e o período histórico de armazenamento.

Elasticidade : Capacidade de adaptação à mudanças dinâmicas das entidades monitoradas, ou seja, os recursos criados e destruidos devem ser corretamente mensurados, adicionados e removidos do sistema de monitoramento.

Adaptabilidade : Adaptação aos vários tipos de recursos, carga de rede e computacional por forma a não ser invasivo afetando o funcionamento global da Nuvem, com consequente degradação da performance de aplicações dos usuários ou excesso de alocação de recursos.

Pontualidade : Eventos detectáveis estão disponíveis a tempo da utilização proposta, evitando latência para a criação de *snapshots* da Nuvem.

Autonomia : Alteração dos sistemas atuais monolíticos para autônomos, que possuam inteligência para inferir e se adaptar a dinamicidade de mudanças na Nuvem.

Compreensividade : Suporte à diferentes tipos de recursos e dados de monitoramento, além de vários fornecedores.

Extensibilidade : O seu suporte é extensível facilmente, como por exemplo através da padronização de APIs para a gama tão variada existente.

Intrusividade : Cuja implementação exige demasiada alteração na Nuvem.

Resiliência : Cuja persistência do serviço entregue é confiável em face à mudanças.

Confiabilidade : Capacidade de executar uma tarefa sob condições específicas durante determinado tempo.

Disponibilidade : Provê os serviços de acordo com a a especificação quando lhe é requisitado.

Acurácia : Provê medidas acuradas, ou seja, o mais próximo da realidade.

3.5 MÉTRICAS DE MONITORAMENTO (MM)

O objetivo das métricas é também fazer gráficos interessantes para entender a capacidade, a performance, e como varia o seu sistema ao longo do tempo. Assim, consequentemente verificar se o sistema funciona como o esperado e tomar uma decisão caso o contrário.

Na CN, podemos ter tanto monitoramento de *alto* como de *baixo* nível, e ambos são necessários, (MEI; CHAN; TSE, 2008). O *monitoramento de alto nível* está relacionado com a informação sobre o *estado da plataforma virtual*. Essas informações são coletadas nas camadas de *MW*, aplicações e usuários por fornecedores ou consumidores através de plataformas e serviços operados pelo provedor ou por terceiros. No caso do *SaaS*, as informações de monitoramento de alto nível é geralmente de maior interesse para o consumidor do que para o provedor. Por outro lado, o *monitoramento de baixo nível* está

relacionado às informações recolhidas pelo fornecedor e, geralmente, não exposto ao Consumidor, e está mais preocupado com o estado da infraestrutura física de toda a Nuvem. No contexto do IaaS, ambos os níveis são de interesse para ambos os consumidores e fornecedores, (ACETO et al., 2013).

Mais precisamente em Spring (2011), para o monitoramento de baixo nível utilitários específicos coletam informações na camada de *HW* (por exemplo, em termos de CPU, memória, temperatura, tensão, carga de trabalho, etc), na camada de SO e na camada de *MW* (por exemplo, bugs e vulnerabilidades de software), na camada de rede (sobre a segurança de toda a infraestrutura através de firewall), e na camada de instalação (a segurança física das instalações envolvidas por meio de monitoramento de ambientes de *datacenter* usando videovigilância e sistemas de autenticação).

Além das camadas de monitoramento, outro fator importante a se definir são as *métricas*. Estas podem ser classificadas consoante a internalidade ou não ao sistema: *Cliente* (externa) ou *Sistemas* (interna). Assim como, tradicionalmente é classificada quanto ao Processamento, *Entrada/Saída (ES)* ou Redes.

3.6 FERRAMENTAS DE MONITORAMENTO (FM)

Em um cenário real, onde o crescimento progressivo do sistema demanda cada vez mais profissionais especializados, infelizmente a tecnologia não acompanha a necessidade deste crescimento. Ainda hoje, numa situação comum com 400 servidores, é necessário despender 15min para iniciar um deles e 1 hora para verificá-los com ferramentas desatualizadas para realidade da Nuvem, como o Cacti, Nagios™, Ganglia© e ferramentas *SaaS* externas.

Em Aceto et al. (2013)além das propriedades para o *MN* é também analisado as principais plataformas *"opensource"* (código-aberto) e comerciais. O âmbito deste trabalho se limita a análise das ferramentas de código-aberto. O autor disponibiliza uma tabela na Figura 12, indicando a *aderência das ferramentas às propriedades por ele elencadas*. Ao analisa-las fica claro que o modelo proposto em Schubert (2014) e consequente implementação de uma ferramenta dedicada ao *MN* é necessária, pois a maioria das ferramentas apresentadas são uma evolução daquilo que era utilizado no *monitoramento em rede (MR)* e, portanto, não atendem todas as propriedades de Aceto et al. (2013) e adaptações necessárias à mudança

de paradigma para CN.

Uma das ferramentas do nosso projeto CloudStack™, que uma plataforma de *gerência de redes (GR)* adaptada a realidade da Nuvem só é alinhada no quesito *Pontualidade*. Assim como outras ferramentas adaptadas, possuem pouca aderência a outras propriedades, destacando-se *elasticidade* e *escalabilidade*. Isto posto, pode-se inferir que as ferramentas de *"opensource"* não apresentam aderência significativa ao Ambiente da Nuvem. Uma das poucas ferramentas que ainda atende outro *requisito funcional*, além das *extensibilidade* e *elasticidade* de Aceto et al. (2013), será descrita melhor em uma subseção a seguir, é o *framework* Sensu®.

Platform	Scalability	Elasticiy	Adaptabilitiy	Timeliness	Autonomicity	Comprehensiveness	Extensibility	Intrusiveness	Resilience	Reliability	Availability	Accuracy
Nagios [104]							✓					
OpenNebula [105]	✓		✓									
CloudStack ZenPack [109]				✓								
Nimbus [110]					✓							
PCMONS [111]							✓					
DARGOS [128]			✓				✓	✓				
Hyperic-HQ [138]	✓					✓						
Sensu [139]		✓					✓					

Figura 12 – Plataformas *opensource* de MN, (ACETO et al., 2013).

3.6.1 Nagios™

Nagios™ é uma plataforma de código-aberto para monitoramento bem conhecida, mas que foi estendido para apoiar o acompanhamento de infraestruturas em Nuvem. Ampliou-se os recursos de Monitoramento para ambas as instâncias: *virtuais* e *serviços de armazenamento*. Graças a tais extensões tem sido adotado para monitorar *Eucalyptus (elastic utility computing architecture)*[12], que é uma plataforma de CN em código-aberto

[12] É um sistema de infra-estrutura para implementação de uma Nuvem Híbrida (NH) do tipo IaaS.

compatível com o EC2 e S3 da Amazon™. Ele também é usado para monitorar OpenStack™ , uma plataforma também de código-aberto para Nuvem IaaS (o SO Ubuntu® adota como solução padrão de Nuvem Privada (NPi) desde o lançamento da sua versão 11.10).

O Nagios é composto por três projetos principais: *Computação*, *Armazenamento de Objetos*, *Serviços de Imagem*.

A principal característica oferecida pelo Nagios é *extensibilidade*, como pode ser visto na Figura 12.

E seus principais problemas são:

- Interface confusa ou com pouca usabilidade;
- Arquitetura Monolítica[13],
- Pouco escalável e muito complicada de adicionar nós dinamicamente.

3.6.2 Zenoss Core®

O Zenoss Core® foi desenvolvido pela empresa comercial, Zenoss Inc., Que foi co-fundada por *Erik Dahl* e *Bill Karpovich* em 2006. Antes de fundar a Zenoss Inc., Dahl iniciou o desenvolvimento de Zenoss em 2002 para atender a uma necessidade da própria empresa. A sua concepção inicial era de uma solução acessível, funcional e fácil de usar para organizações de todos os tamanhos, (BADGER, 2008).

O Zenoss Core® é um aplicativo web que se instala a um servidor central na Rede e usa o servidor de aplicações *Zope* - servidor de aplicações web de código aberto. Seu nome significa Z Object Publishing Environment. Muitas tarefas de adminitração de um servidor Zope podem ser realizadas através de uma interface web. É utilizado no Laboratório de Redes e Gerência para o *monitoramento da Nuvem (MN)* e, é escrito em Python e baseado em Linux. Sua configuração é bem simplificada, multiplataforma e não exige grandes conhecimentos, mas utiliza apenas VMware® Player ou o VMware® Server. Os instaladores nativos do Linux continuam a melhorar e apoiar uma ampla gama de distribuições, o que significa que as habilidades necessárias para instalar Linux Zenoss Core® nativamente continua a diminuir. A partir da versão 2.2, terá a opção de instaladores

[13] É a arquitetura de SO mais comum e antiga, onde cada componente do SO está contido no núcleo do sistema.

point and click, sendo atualmente instalado a partir do código. Os administradores acessam o Zenoss Core® através de uma interface web que nos permite fazer:

- Gerenciamento de Dispositivos;
- Disponibilidade e Monitoramento de Desempenho,
- Gestão de Eventos,
- Sistema de Geração de Relatório,
- Gerenciamento de Usuários e Alertas.

3.6.3 CloudStack™

CloudStack™ é um software de código-aberto escrito em Java™, projetado para implantar e gerenciar grandes redes de *MVs*, como uma plataforma de Nuvem altamente disponível e escalável. É utilizado no Laboratório de Redes e Gerência para o *gerênciamento da Nuvem (GN)* e, atualmente ele suporta os hipervisores mais populares (por exemplo, VMware®, Oracle™ VM, KVM, XenServer™ e Xen Cloud Platform) e oferece três maneiras de Gerenciar Ambientes de Nuvem:

- Uma interface fácil de usar web;
- Uma ferramenta de linha de comando, CloudMonkey©[14],
- Uma API RESTful[15] completa.

3.6.4 ZenPack Zenoss inc.™

Com a finalidade de monitorar dispositivos *físicos* e *virtuais*, uma extensão chamada ZenPack pode ser usado. Ele gerencia os eventos e alertas, como pode ser visto na Figura 13, entretanto, como deveria ser fornece os parâmetros relacionados com a *memória*, *CPU* e *armazenamento*, assim como para a *Rede*, um exemplo de gráficos de monitoramento desta ferramenta pode ser visto na 14. Assim como o CloudStack™ principal característica oferecida pelo ZenPack é *Pontualidade*, provavelmente entre as propriedade definidas na

[14] Interface de linha de comando (CLI) para CloudStack usada tanto como um shell interativo ou ferramenta de linha de comando que simplifica a configuração e gerenciamento.

[15] Os sistemas que seguem os princípios REST são freqüentemente chamados de RESTful

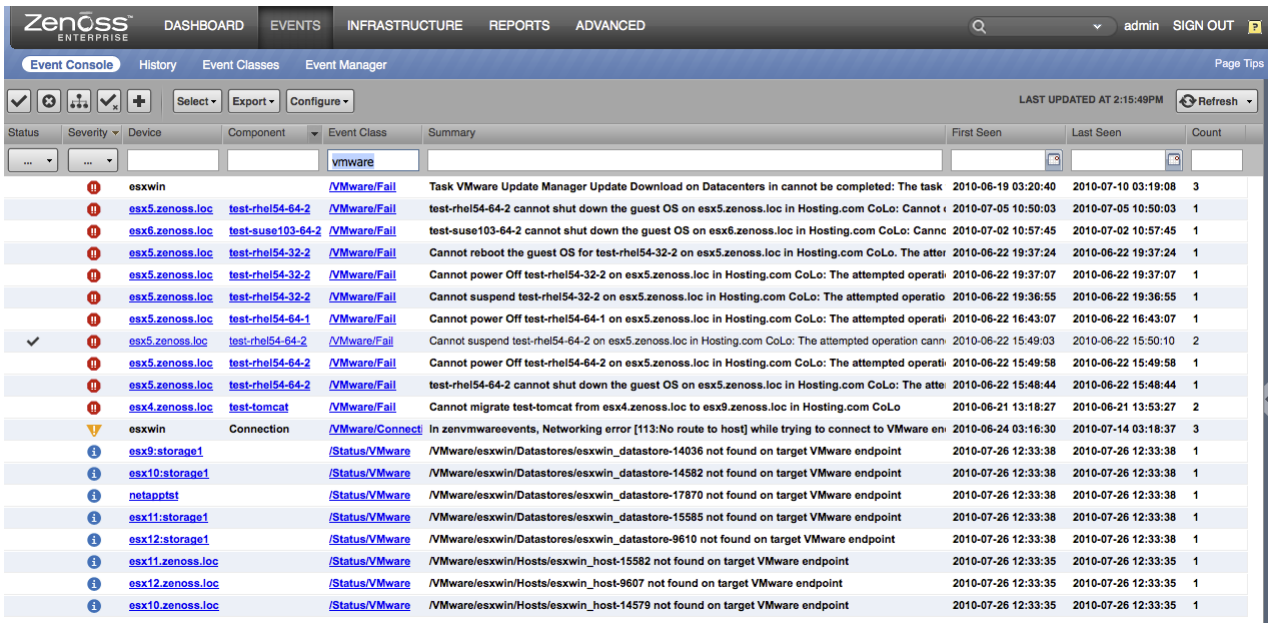

Figura 13 – Visão (janela) onde ficam registrados os eventos e alertas CloudStack™ no Zenoss™, (WIKIZENOSS, 2014).

taxonomia já citada, a mais interessante para monitoramento de Nuvens, principalmente quando se trata de manipulação de dados em séries temporais como vai ser visto adiante e, consequentemente, provavelmente mais adequado para o modelo de monitoramento autônomo e cognitivo proposto por Schubert (2014), pois se adequa as situações é necessário *tomadas de decisão (TD)*.

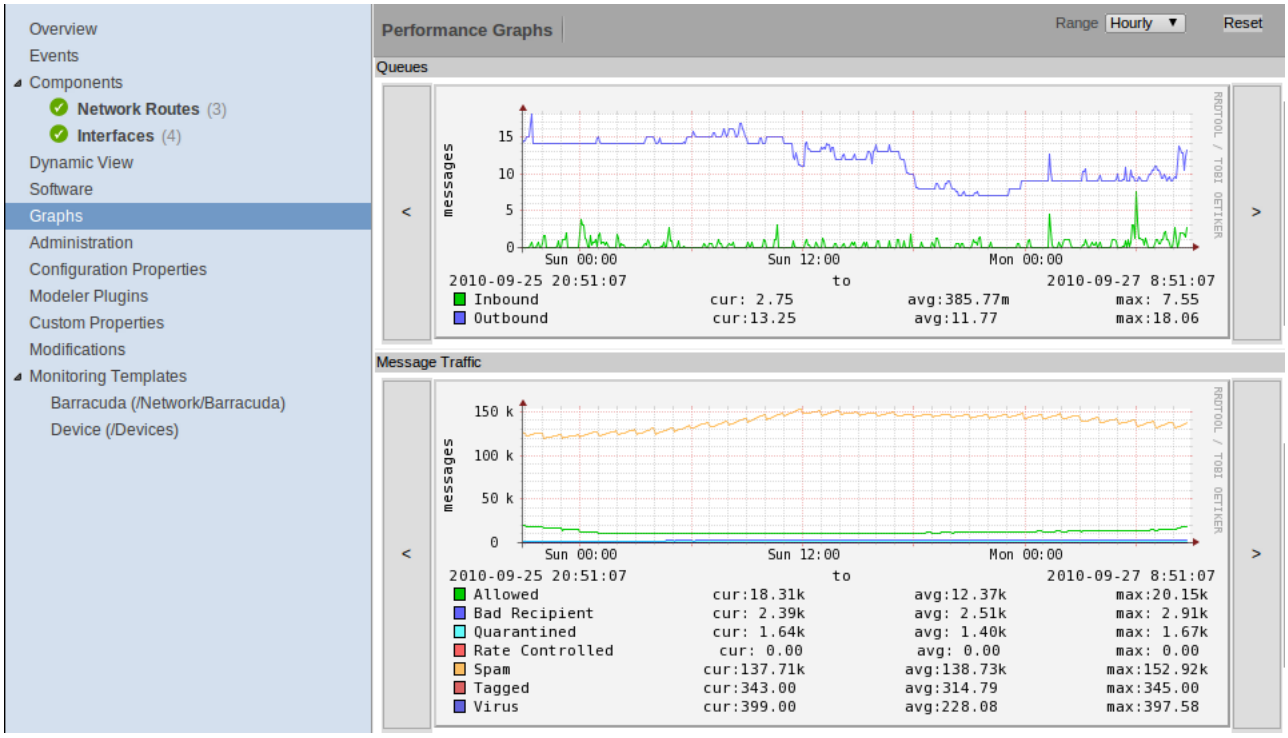

Figura 14 – Métricas monitoradas pelo ZenPack Zenoss inc.™, (WIKIZENOSS, 2014).

A maneira mais fácil de começar a monitorar a Nuvem é navegar para a página de

Infraestrutura do Zenoss™ e adicionar o CloudStack™, e essa a infraestrutura utilizada no LRG. Uma vez adicionado um ao outro com sucesso, é possível começar a ver as métricas a seguir disponíveis para toda a Nuvem no Zenoss Core®. Esses números são agregados a partir de todas as *Zonas, Pods, Clusters, Hosts*:

- *IPs públicos:* Total e Usado;
- *IPs privados:* Total e Usado,
- *Memória:* Total (com e sem excesso de provisionamento), atribuição e utilização,
- *CPU:* Total (com e sem excesso de provisionamento), atribuição e utilização,
- *Armazenamento primário:* Total (com e sem excesso de provisionamento), atribuição e utilização,
- *Armazenamento secundário:* Total e Usado,
- *Rede:* Leitura e Escrita.

A mesma lista de métricas estão disponíveis para cada *Zona*. As mesmas métricas, com exceção de IPs públicos e armazenamento secundário também estão disponíveis para cada *Pod*. As seguintes métricas estão disponíveis agregadas para cada grupo e para cada *host*.

- *Memória:* Total e Usado;
- *CPU:* Cores Total (com e sem excesso de provisionamento), atribuídos, usados,
- *Rede:* Leitura e Escrita.

3.6.5 Sensu™

Projetado para superar os limites das plataformas tradicionais de monitoramento em ambientes de Nuvem, O Sensu™ é baseado em RabbitMQ, um *MW* orientado a mensagem que inclui um servidor de monitoramento, os agentes independentes de plataforma e um painel de controle baseado na web. Ele aproveita se utiliza da *advanced message queing protocol (AMQP)* - protocolo padrão para publicação, enfileiramento, armazenamento e consumo de mensagens. O produtor de mensagens utiliza uma API para publicar as

mensagens no servidor. O servidor/broker irá colocar as mensagens em filas e armazená-las enquanto estes não são consumidos. Por fim, uma aplicação irá consumir as mensagens removendo-as das filas - para realizar um processamento escalável e comunicação segura implementando uma API JSON[16] REST para recuperação de dados. A plataforma está voltada principalmente para as propriedades de *extensibilidade* e *elasticidade*, o que cobre mais propriedades elencadas por Aceto et al. (2013), o que não significa que seja a ferramenta mais adequada para este *modelo de dados(MD)*, uma vez que o *CloudStack Zenpack*^{TM} atende à propriedade de *timeliness (pontualidade)*, mais importante para o MN. Mas, suas principais características são:

- Por ser orientada a mensagens utiliza JSON em todo lado;
- Pode utilizar as verificações do Nagios^{TM},
- Projetado para ser orientado a gerência de configurações da Nuvem,
- Suporte à topologias dinâmicas, ou seja, adapta-se a mudança na topologia constante da Nuvem.

[16] É um acrônimo para JavaScript Object Notation, é um formato leve para intercâmbio de dados computacionais.

4 BIG DATA, ARMAZENAMENTO DE DADOS E TECNOLOGIAS

4.1 BIG DATA

Um aspecto crescente da *computação em nuvem (CN)* é gerir grande volume de dados, *big data*. No cenário atual, o termo *big data* é descrito como um fenômeno que se refere à prática de coleta e processamento de grandes conjuntos de muitos dados, os sistemas associados e algoritmos usados para analisar *massas de dados (MD)*. Três características bem reconhecidas de dados grandes são *variedade*, *volume* e *velocidade* (3Vs) de geração de dados. O crescimento constante das mídias sociais e dispositivos móveis tem levado a um aumento das fontes de tráfego de saída, iniciando o conceito de *big data*, o que impõe desafios significativos na *CN*.

Os recentes estudos mostram que quanto mais pessoas se juntarem aos sites de mídia social hospedados em Nuvens, a análise dos dados se torna mais difícil e quase impossível de ser feita. Outros aspectos estudados são: como que através da migração de *MV* e, copiar e salvar o estado atual, *snapshot* da Nuvem, interfere no desempenho de transferência de dados da mesma. Além disso, os diferentes tipos de dados provenientes de *dispositivos móveis* torna a compreensão de dados um problema desafiador devido à: *multi-modalidade*, *grande volume*, *dinamicidade*, *várias fontes* e *qualidade imprevisível*. O monitoramento contínuo dos fluxos de dados multimodais coletados de fontes heterogêneas requerem ferramentas de monitoramento para lidar com o gerenciamento deste grande volume de dados.

Existem várias ofertas de armazenamento de código aberto em NoSQL. Desde a publicação da tecnologia BigTable™, muitas opções têm surgido incluindo HBase™, Hypertable™, e Cassandra™. Estas tecnologias estão atualmente destacadas em empresas como Facebook™, Reddit™, e Baidu™.

Muitos recursos relacionais comuns foram removidos e armazenados em NoSQL para atingir o conjunto de recursos de *alta disponibilidade*, *alto rendimento* e *tolerância a falhas*. Esses recursos incluem SQL e suporte a transações que podem causar *gargalos*, *tempos de resposta de alta latência*, e até mesmo *falha no sistema* ao lidar com *MD*. Como os armazenamentos de dados NoSQL não tem uma linguagem de consulta totalmente

expressiva embutida, tecnologias e adaptadores adicionais foram adicionados para analisar os dados. A tecnologia predominante é MapReduce™, que se tornou popular devido sua utilização no Google™ desde 2004.

MapReduce™ e outras implementações como o Hadoop™, são capazes de executar *mapeadores* e *redutores* em um conjunto de dados que se estende por várias máquinas ao manusear automaticamente falhas. Ferramentas de nível superior, como Pig© e Hive© foram criadas para abstrair a necessidade de escrever mapeadores e redutores e fornecer uma interface SQL familiar para acessar os dados. Essas ferramentas podem ter grandes volumes de dados e facilitar a consulta analiticamente.

Desenvolvedores agora enfrentam múltiplos desafios ao selecionar um armazenamento de dado NoSQL para se confiar. Qualquer tecnologia em particular tem uma API específica e, portanto, migrar de uma tecnologia para outra requer um esforço portabilidade caro.

Além disso, como estas tecnologias estão em sua infância em comparação com as ofertas SQL, existem constantes atualizações do *SW* e isso requer atualizações para obter correções de *bugs*, novos recursos e melhorias de desempenho.

As empresas também enfrentam desafios relacionados com a infinidade de opções de tecnologias NoSQL. Em particular, não há nenhuma maneira fácil para os desenvolvedores para *comparar* e *constrastar* as diferentes ofertas sem ter que se tornar um *especialista* em um conjunto de armazenamentos de dados; Não existe uma maneira simples de migrar de um sistema de armazenamento para outro, muito menos, os desenvolvedores são incentivados a desenvolver tais ferramentas.

Aplicações escritas para serem *altamente escaláveis* podem não se encaixar em todos cenários de plataformas de armazenamento de dados existentes, e as diferentes aplicações podem obter um melhor desempenho usando um armazenamento de dados específico para sua arquitetura. Além disso, partes de um aplicativo podem exigir recursos como transações, em detrimento da escala.

Pensando em todas estas questões e relacionando ao cenário que temos da ACAMN, definiu-se a criação de uma BD NoSQL para dar suporte às métricas da Nuvem utilizando o *Gerenciador da Nuvem (GN)*: CloudStack™ e, *monitorador da Nuvem (MN)*: Zenoss™.

Apesar de guardar um menor volume de dados, ou seja, uma *MD* não tão significativa como a gerada por diversas aplicações heterogêneas, é o suficiente para provavelmente gerar problemas de *tempo de resposta de alta latência*, *escalabilidade* em grandes *clusters* da Nuvem, etc. Principalmente quando se trata de monitoramento em tempo-real onde o volume de dados é maior. Nas sessões que se seguem é feita a descrição dos sistemas de BDs Relacionais (BDRs), NoSQL e BD Temporal, também, é discutida a terminologia, tecnologias, propriedades e os conceitos usados nessas BDs.

4.2 BANCO DE DADOS (BD)

Os primeiros desenhos de BDs e implementações foram baseados no uso de *listas ligadas* para criar relações entre os dados e encontrar dados específicos. Eram uma forma de navegação baseada nestas listas, posteriormente surgiram os BDs relacionais, depois orientados a objetos e, no final da década de 2000 o NoSQL surgiu e tornou-se uma tendência popular, (BERG; SEYMOUR; GOEL, 2013).

Os *bancos de dados(BD)* foram criados a fim de satisfazer a necessidade de armazenar e recuperar dados de forma organizada. Desde a sua criação nos idos de 1960 surgiram diversos usando sua própria representação de dados e tecnologia para lidar com transações.

Os BDs são definidos como coleções organizadas de dados, (BEYNON-DAVIES, 2004). Embora quando se utiliza o termo *banco de dados (BD)* referindo-se a todos sistema, este apenas se refere à coleta e os dados. O sistema que controla os dados, transações, problemas ou qualquer outro aspecto do BD é o *sistema de gerenciamento de banco de dados (SGBD)*.

Apesar de BDs NoSQL serem relativamente novos em comparação com outros tipos que se tornaram populares devido à sua capacidade de lidar de forma muito rápida com dados não relacionados e não estruturados, principalmente porque eles não precisam de um esquema fixo e usam fortemente *metadados* [1], a fim de alcançar um desempenho rápido. Os BDRs vieram à tona em meados da década de 1970 e têm bem sucedidos até o final da última década. Os conceitos essenciais envolvidos em BDR foram estabelecidas por *Edgar Codd* , a fim de superar as desvantagens das implementações de *listas ligadas*

[1] São dados sobre outros dados, geralmente uma informação inteligível, facilitam o entendimento dos relacionamentos e a utilidade das informações dos dados.

anteriores em BDs.

4.3 BANCO DE DADOS RELACIONAIS (BDR)

Banco de dados relacionais (BDR) usam a noção de BDs separados em *tabelas* onde cada coluna representa um *campo* e cada linha representa um textbfregistro. As tabelas podem ser relacionadas ou ligadas umas com as outras, com a utilização de *chaves estrangeiras* ou colunas comuns. Em um nível abstrato as tabelas representam *entidades*, como, por exemplo, usuários, clientes ou fornecedores. Essa abstração é útil ao projetar o *esquema* de BDs como objetos do mundo real numa forma de mapeado para o BD, além disso é mapeado também as relações entre eles. Um exemplo de um *esquema* de BD pode ser visualizados por meio de diagramas, como na Figura 15.

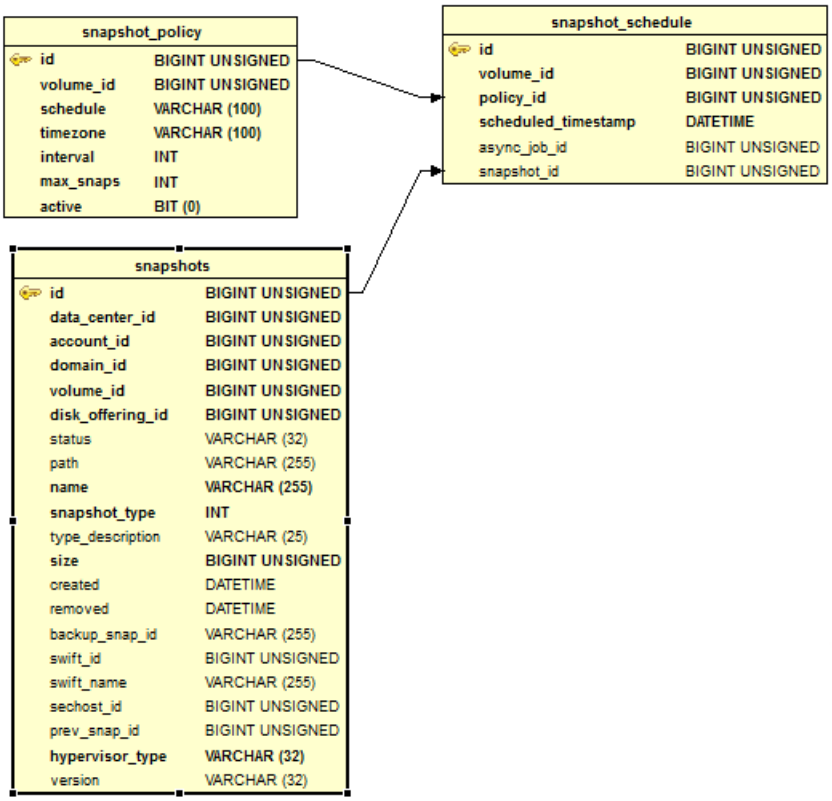

Figura 15 – Tabelas de snapshots (instantâneos) do CloudStack^{TM}, (KUMAR, 2013).

Um aspecto importante do projeto de BDRs é a *normalização* do esquema. Que é feita resumidamente em três passos:

1ª Forma Normal (1NF): Elimina-se os grupos de dados repetidos, criando uma nova tabela para cada grupo de dados relacionados que é identificado por

uma *chave primária*.

2ª Forma Normal (2NF): Se um conjunto de valores são os mesmos para vários *registros*, é necessário movê-los para uma nova tabela e ligar as duas tabelas com uma *chave estrangeira*.

3ª Forma Normal (3FN): Os campos que não dependem da chave primária de uma tabela deve ser removido e, se necessário, ser colocado em outra tabela.

Também há condições para um *esquema de BDs*, onde a *1NF* deve ser feita para depois satisfazer a *2NF* e o mesmo se aplica respectivamente para *3NF*. Embora existam outras formas de *"schema"* (esquema), atendendo as 3NFs já é considerado *normalizado*.

Quando o modelo relacional adotou a linguagem *(structured query language (SQL)* que era a linguagem de consulta de dados mais utilizada no mundo e disponível para praticamente todas as plataformas de HW, desde *mainframes* até computadores portáteis, este modelo consolidou-se como o modelo mais famoso e utilizado, tornando-se padrão para a maioria dos SGBDs, como nos mais utilizados atualmente o Oracle™, SQL Server™, PostgreSQL™ e o MySQL®, (ELMASRI; NAVATHE, 1999).

4.3.1 Structured Query Language (SQL)

A versão original do SQL foi desenvolvida pela IBM™. Essa linguagem, era originalmente chamada de *Sequel*, foi implementada como parte do projeto do Sistema R no inicio dos anos 70. Desde então a linguagem *Sequel* foi evoluindo e seu nome mudado para SQL, (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

Emerson, Darnovsky e Bowman (1989), define a linguagem SQL como sendo tanto uma linguagem de consulta interativa como uma linguagem de programação de BD. E tem como função facilitar o acesso à informações armazenadas em BDRs, através de consultas, atualizações, e manipulações de dados.

Segundo (OLIVEIRA, 2002)a linguagem SQL é composta por quatro grupos:

Data Manipulation Language (DML):

- Permite a manipulação dos dados armazenados no BD. Possui os comandos: DELETE, INSERT e UPDATE.

Data Definition Language(DDL):

- Permite a criação dos componentes do BD. Principais comandos são: CREATE TABLE, ALTER TABLE, DROP TABLE,CREATE INDEX, ALTER INDEX e DROP INDEX.

Data Control Language(DCL):

- Provê a segurança interna do BD. Formado pelos comandos: CREATE USER, ALTER USER, GRANT, REVOKE e CREATE SCHEMA.

Data Query Language(DQL):

- É formado apenas pelo comando SELECT. E tem a função de extrair dados no BD.

4.3.1.1 MySQL®

Silberschatz, Korth e Sudarshan (2006), definem *SGBD* como uma coleção de dados inter relacionados conjuntamente com programas utilitários com a função de acesso e manipulação dos mesmos. O *SGBD* tem como função proporcionar um ambiente conveniente e eficiente para recuperação e armazenamento das informações.

O *MySQL*® foi um projeto que teve início em 1984, cujo principal objetivo era de garantir aos seus usuários a liberdade de compartilhar e alterar seus aplicativos, e assim tornando-os livres. É um servidor multi-thread, que atende a vários usuários ao mesmo tempo, para isso são criados múltiplos processos de execução em um único processo. Suas tarefas são executadas em backend, isto é, não são controladas pelo usuário.

O *MySQL*® é um gerenciador utilizado pelas principais ferramentas de monitoramento (FM) atuais, inclusive as utilizadas na Nuvem do LRG, foi construído em múltiplas camadas que podem ser classificadas basicamente em, (DYER, 2008):

Camada de Aplicação: representa a interface de usuário, que é como o servidor MySQL interage com seus usuários.

- *Interface de Consulta:* São as instruções DML;
- *Cliente:* Pessoas ou programas que utilizam a interface do servidor MySQL;

- *Interface de Administrador:* Representa os utilitários utilizados para administração do servidor MySQL.

Camada Lógica: Processador de Consultas, Gerenciamento de Transações, Gerenciamento de Recuperação.

Camada Física: forma como o MySQL mantém o armazenamento físico dos dados em arquivos. Além dos dados, dicionários, índices, históricos (logs), estatísticas e todo o controle dos recursos de memória são controlados pela camada física, (DYER, 2008).

- Gerenciador de Recursos;
- Gerenciador de Buffers,
- Gerenciador de Armazenamento.

A arquitetura do *MySQL*® é útil para uma grande variedade de objetivos. Ao mesmo tempo, *MySQL*® pode potencializar depósitos de dados, indexação de conteúdo e software de distribuições, sistemas redundantes altamente disponíveis, processamento de transação on-line e muito mais, (SCHWARTZ et al., 2008).

4.3.1.1.1 Tipos de Tabelas

A tabela ou ferramenta de armazenagem é considerada de maior importância neste contexto, pois sua função é o ponto chave da existência de um BD. Sua estrutura é formada de linhas que são os registros, os quais se referem às ocorrências desses objetos no mundo real, e as colunas ou campos, que são os seus atributos, (MYSQL, 2014).

O *MySQL*® suporta dois tipos diferentes de conjunto de tabelas: *tabelas seguras com transação (TST)* e tabelas não seguras com transação (TNST), o CloudStack™ e o Zenoss™ armazenam os dados em *InnoDB* que é uma tabela *TST*.

4.3.1.1.2 Tabelas innoDB

Segundo MYSQL (2014), *InnoDB* é o tipo de tabela que provê um mecanismo de armazenamento seguro de transações com confirmação da transação, *rollback*[2] e recupera-

[2] Mecanismo de reversão de transação em banco de dados (BD).

ção em caso de falhas. Este tipo de tabela é utilizada pelas ferramentas de monitoramento principalmente, e faz bloqueio em nível de registro e também fornece leituras sem bloqueios. Estes recursos, fazem com que haja um aumento do desempenho e a concorrência de multiusuários. O InnoDB foi desenvolvido para obter o máximo de desempenho ao processar um grande volume de dados. Há relatos de sites que utilizam InnoDB em base com mais de 1 *TeraBytes* de dados, processando uma carga média de 800 inserções/atualizações por segundo, (MYSQL, 2014).

As principais características da *InnoDB* são: permite chaves estrangeiras, bloqueios a nível de linha, versionamento múltiplo, agrupamento pela chave primária, todos os índices contém as colunas da chave primária, cache otimizado, índices descompactados e bloqueio de incrementação. É considerado um tipo de *tabela lenta*, pois seus dados são armazenados diretamente no disco, não deixando em cache de escrita da *Random Access Memory (RAM)*[3].

Lento, porém proporcionalmente seguro, o que para aplicações altamente escaláveis como no caso de um Ambiente em Nuvem não se torna muito interessante, (SCHWARTZ et al., 2008). Na seção de experimento do próximo capítulo, será possível comprovar esta afirmação.

4.4 PROPRIEDADES ACID

Um critério importante para garantir a *confiabilidade* das transações dos *BDRs* é a aderencia às propriedades *ACID*: Atomicidade, Consistência, Isolamento, Durabilidade, respectivamente, (BEYNON-DAVIES, 2004).

Atomicidade: Todas as partes de uma transação devem ser concluídas ou nenhuma.

Consistência: A integridade dos dados é preservada por todas as transações. O BD não é deixado em um estado inválido depois de uma transação.

Isolamento: Uma transação deve ser executada isolada, a fim de garantir que

[3] Memória de acesso aleatório, é um tipo de memória que permite a leitura e a escrita, utilizada como memória primária em sistemas eletrônicos digitais.

qualquer inconsistência nos dados envolvidos não afeta outras transações.

Durabilidade: As alterações feitas por uma transação concluída deve ser preservada ou em outras palavras, ser durável.

4.5 TEOREMA CAP

Existem muitas motivações para utilização dos *BDs NoSQL*, como por exemplo facilitar alterações de *esquema*, melhorar o desempenho e simplificar a replicação para ter a tão sonhada escalabilidade linear.[4]

Entretanto, todos os benefícios não vem sem um custo, comparado com os BDs tradicionais perde-se alguma funcionalidade ou garantia para se ganhar outra. O *"tradeoff"* (desvantagem) arquitetural é descrito no bem conhecido *teorema CAP*, o qual explica que em qualquer *Sistema Distribuído (SD) statefull*[5] é preciso escolher entre *C - consistência forte*, *A - alta disponibilidade* e *P - tolerância a particionamento dos dados na rede*. Segundo o *teorema CAP*, entre as três propriedades, *somente duas* podem ser garantidas ao mesmo tempo, (MESSINGER, 2013).

Para (MESSINGER, 2013), sistemas que precisam da *consistência forte* e *tolerância a particionamento (CP)*, é necessário abrir a mão da disponibilidade um pouco. Pode acontecer, caso haja particionamento e o sistema não entre em consenso, que uma escrita seja rejeitada. Claro que os sistemas tentam evitar isso ao máximo, tanto que não costuma existir, por exemplo, uma transação distribuída e sim um protocolo de consensos para garantir a consistência forte. Exemplos desses sistemas *CP* são BigTable™, HBase™ ou MongoDB™ entre vários outros.

Os sistemas com *consistência forte* e *alta disponibilidade* (CA) (alta disponibilidade de um nó apenas) não sabem lidar com a possível falha de uma partição. Caso ocorra, sistema inteiro pode ficar indisponível até o membro do *clusters* voltar. Exemplos disso são algumas configurações clássicas de *BDRs*, como neste trabalho o *MySQL*®.

É importante mencionar que para o desenvolvedor não haverá tantas diferenças

[4] É uma característica desejável em todo o sistema, em uma rede ou em um processo, que indica sua habilidade de manipular uma porção crescente de trabalho de forma uniforme e linear, ou estar preparado para crescer.

[5] Que normalmente mantém uma cópia do estado atual do sistema.

entre *CA* ou *CP*. Sempre terá consistência forte, no entanto, um sistema fica indisponível (CA) quando há particionamento – pois tem apenas alta disponibilidade por nó – e o outro sistema (CP) tente chegar à um consenso se aceita uma escrita ou não, que no pior dos casos também pode significar a indisponibilidade para uma parte dos dados. Partindo desse raciocínio é possível perceber que a *consistência* e *disponibilidade* são extremos quando há *particionamento*, como se pode ver no Diagrama da Figura 16.

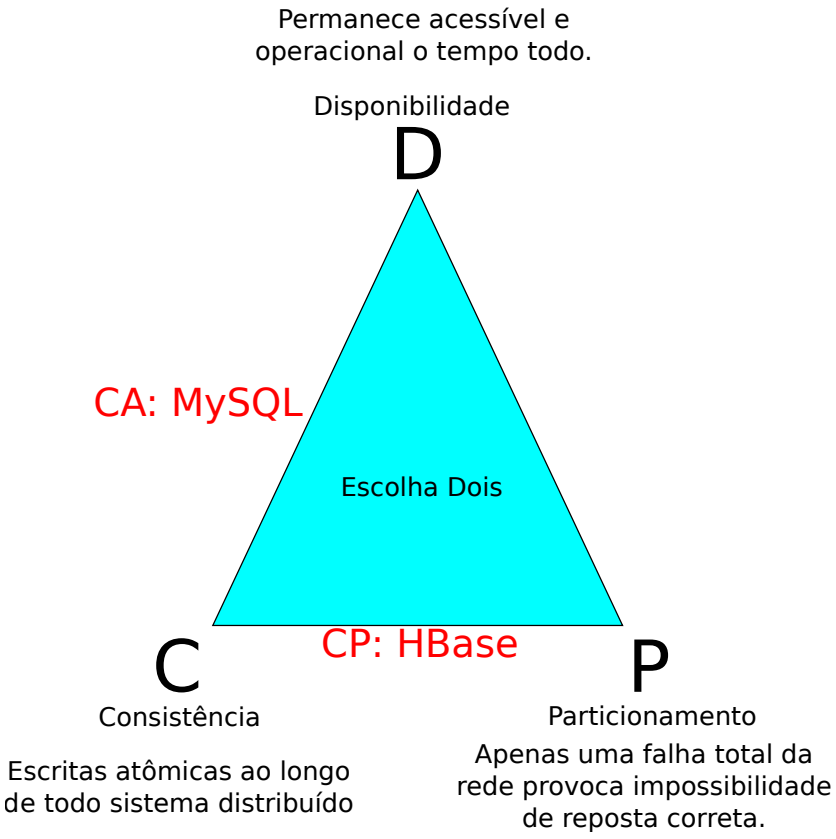

Figura 16 – Diagrama do Teorema CAP, autoria nossa.

O *teorema CAP* não responde o que acontece *sem particionamento*. A primeira resposta poderia ser: claro que vai ser consistente já que ninguém gosta de lidar com dados desatualizados. Mas olhando para os sistemas NoSQL nem sempre isso é verdade. Existem sistemas que sempre são *eventualmente consistentes*. Há mais um motivo porque poderia fazer sentido sacrificar a consistência: O tempo da resposta ou a latência. Da mesma maneira que um sistema *"offline"* pode custar caro, um sistema lento também pode. Por isso pode fazer sentido abrir a mão da consistência para diminuir a latência.

4.6 BANCO DE DADOS NÃO APENAS SQL (NOSQL)

BDs NoSQL (Not Only SQL - Não Somente SQL) começaram a ganhar notabilidade em 2000, quando empresas começaram a investir e pesquisar mais sobre BDs distribuídos, (SADALAGE; FOWLER, 2012). Por isso, este tipo de BD cresceu e incluiu muitos subtipos cada mais adequados para os conjuntos de dados específicos do que outros. As categorias de base de dados mais comuns NoSQL são os seguintes:

Documentos: A noção de *documentos* é o conceito central sendo o equivalente de registros em BDR e *coleções* sendo semelhante às tabelas.

Chave-Valor: Os dados são armazenados como valores com uma chave atribuída a cada valor semelhante as *hash-tables*. Também de acordo com o BD, pode ser que uma chave pode ter uma coleção de valores,(SADALAGE; FOWLER, 2012).

Grafos: Como a teoria dos grafos a noção de *nós* e *arestas* é o conceito principal deste BD. Nós correspondem a *entidades* como um usuário ou um registro de música e arestas representam as *relações* entre os nós.

4.6.1 SQL x NoSQL

O primeiro fator importante que diferencia *BDs NoSQL* dos BDRs é a utilização de *adjacência livre de índice*, o que quer dizer que cada elemento contém um apontador para o elemento adjacente e não necessita da indexação de cada elemento, (STONEBRAKER, 2010).

Outro aspecto importante das *BDs NoSQL* é que eles não têm qualquer esquema predefinido, os registros podem ter *diferentes campos*, se necessário, isto pode ser referido como um *esquema dinâmico*. Muitos também suportam a *replicação*, que é a opção de ter réplicas de um servidor, isso proporciona confiabilidade, como no caso de um ficar offline, a réplica se tornaria o servidor primário. Todos os servidores executam as mesmas operações e sincronizam seus dados a fim de eliminar eventuais erros.

Por último, em relaçao aos BDRs, os *BDs NoSQL* não garantem plenamente as propriedades *ACID*. A falta de garantias *ACID* é atribuída à sua arquitetura de implantação que tipicamente envolve ter vários nós de modo a conseguir *escalabilidade horizontal* e recuperação em caso de *failover*. Esta implantação também conhecida como *replicação*, cria problemas com sincronização que pode resultar em um nó secundário se tornar primário, mas não tem uma cópia atualizada dos dados.

Os *BDs NoSQL*, além de utilizar uma *application programming interface (API)* ou linguagem de consulta para acessar e modificar os dados, também pode usar o método de MapReduce™, que é usado para executar uma função específica em um conjunto de dados inteiro e recuperar apenas o resultado.

4.6.2 HBase™

Projeto Apache™ baseado no Google BigTable™, é o principal banco de dados utilizado neste trabalho, ele possui uma abordagem de replicação dos dados que envolve um cluster mestre que contém todos os servidores regionais chamados *HRegionServers*. Estes por sua vez efetuam chamadas síncronas aos *clusters* escravos, que irão escrever os dados inseridos ou atualizados paralelamente. O HBase utiliza o *Hadoop Distributed File System (HDFS)*, uma implementação derivada do *Google File System (GFS)*, que possibilita ao HBase™ desfrutar das vantagens de armazenamento já desenvolvidas pela própria Apache™.

Pelo fato de também ser um gerenciador de BDs NoSQL baseado em colunas, o HBase™ possui muitas características em comum com o Cassandra™ por parte da organização, mas também possui suas peculiaridades pelo fato da equipe da Apache™ ter um foco muito diferente do Facebook™ na implementação de seus produtos. Dentre as principais características do HBase estão:

- Escalabilidade linear e modular;
- Consistência nas leituras e escritas;
- Fragmentação de tabelas automática e configurável;
- Automático suporte a falhas entre servidores;

- Possui extensões que tornam fácil a utilização de programas Java para configurar o acesso dos clientes;
- Cache de bloco e filtros para consultas em tempo-real.

HBase™ suporta dados não estruturados e parcialmente estruturados. Para fazer isso, os dados são organizados em famílias de colunas. Um registro individual, chamado de célula no HBase™, é endereçado com uma combinação de *row key*, *column family*, *cell qualifier*, e *time stamp*. Em oposição aos BDRs, no qual é necessário definir bem sua tabela com antecedência, com o HBase™ é possível simplesmente nomear uma família de colunas e então permitir que a célula se qualifique para ser determinada em um certo tempo de execução. Isso permite que você seja bastante flexível e suporta uma abordagem ágil de desenvolvimento.

4.6.2.1 Entidades Lógicas e Coordenadas

De acordo com Dimiduk et al. (2012), as *entidades lógicas* e *coordenadas físicas* em um *esquema* Apache HBase™ são como se segue:

Tabela: Na Figura 17 é representada uma Tabela em Apache HBase™, que é a forma como os dados são armazenados neste banco de dados. Os nomes das tabelas são *strings* (conjunto de caracteres) e compostos por caractéres seguros para o uso em um caminho de sistema de arquivos.

•

Chave da Linha: Dentro de uma tabela, os dados são armazenados de acordo com a sua linha. As linhas são identificados exclusivamente por seus *row-key (chave da linha)*, também em destaque na Figura 17, que não tem um tipo de dados e são sempre tratados como um *array de bytes[]*.

Coluna Família: Os dados dentro de uma linha é agrupado por colunas família. As colunas família, como é indicado na Figura 18 também impactam na disposição física dos dados armazenados no HBase™. Por esta

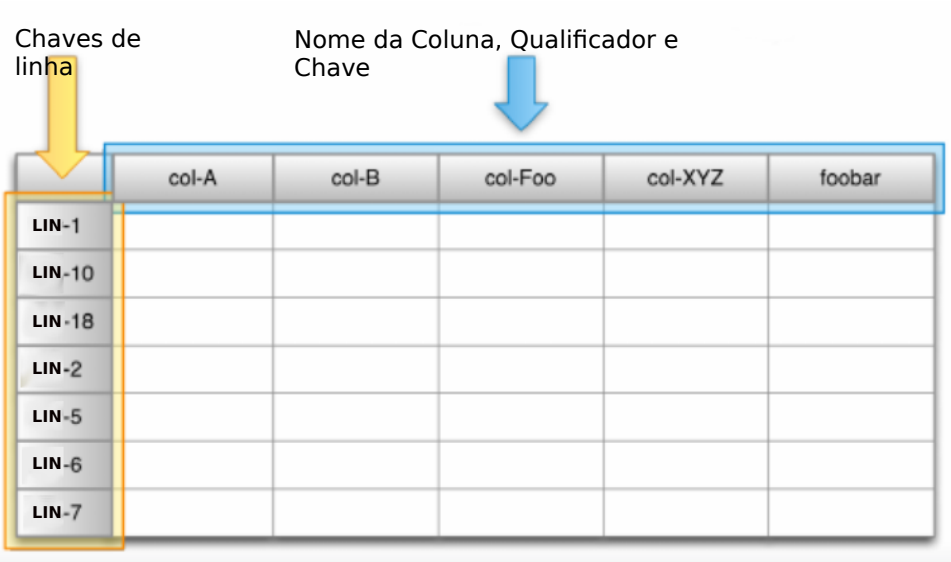

Figura 17 – Entidades de uma tabela HBase™, (GEORGE, 2011).

razão, devem ser definidos antecipadamente e não são facilmente modificados. Cada linha da tabela tem as mesmas colunas família, embora em uma linha não seja necessário armazenar dados em todas as suas famílias. Nomes da família são *strings* e composto por caracteres seguros para o uso em um caminho de sistema de arquivos, utilizado basicamente como um “namespace” e, por essa razão, devem ter o nome o mais curto possível pois são apenas uma referência à um conjunto de colunas e não devem ocupar muitos “bytes”.

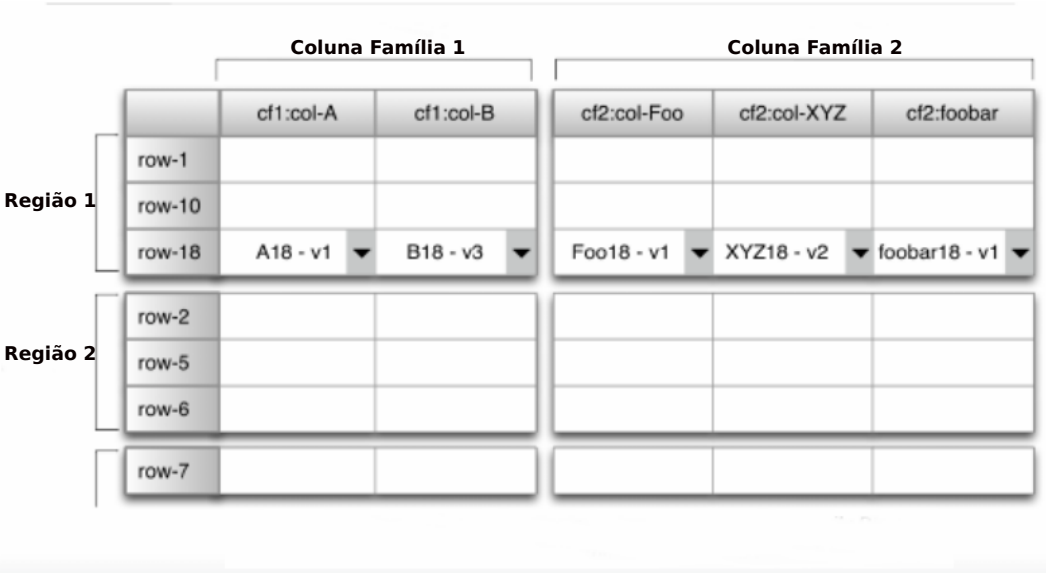

Figura 18 – Coordenadas físicas completas para a célula HBase™, (GEORGE, 2011).

Qualificador da Coluna: Os dados dentro de uma coluna família são referenciados através do

seu qualificador de coluna. Os qualificadores de coluna não precisam ser especificados com antecedência e não precisam ser consistentes entre linhas. Como as *row-keys*, não tem um tipo de dados e são sempre tratados como *array de bytes[]*.

Célula: Uma combinação de *row-key*, coluna família e qualificador da coluna identifica exclusivamente uma célula. Os dados armazenados em uma célula são tratados como o valor dessa célula. Os valores também não tem um tipo de dados e são sempre um *array de bytes[]*.

Versão: Os valores dentro de uma célula são versões. As versões são identificadas por seu *timestamp*, do tipo *long*. Quando uma versão não é especificada, a hora atual é usada como a base para a operação. O número de versões valor da célula retidos pelo HBase™ é configurado através da coluna família. O número padrão de versões nas células é *(3) três*, o versionamento é melhor indicado na Figura 19 .

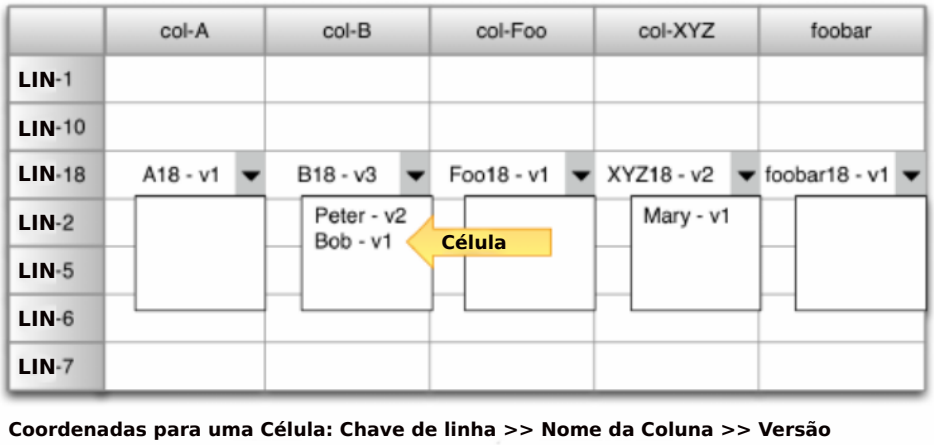

Figura 19 – Coordenadas físicas para a célula HBase™, com versionamento, (GEORGE, 2011).

Estes seis conceitos formam a base da HBase™. Eles estão expostos ao usuário através da visão lógica apresentada pela *API*. Eles são os blocos de construção em que a implementação gerencia dados fisicamente no disco. Um valor de dados único no HBase é acessado por meio de suas coordenadas, esquematizadas na Figura 19. As coordenadas completas, Figura 18 para um valor são *row-keys*, coluna família, qualificador da coluna, e a versão.

No *modelo de dados* para o HBase™, o *número da versão* também faz parte das coordenadas de um conjunto de dados. Pode-se pensar em um *BDR* com o armazenamento de uma quantidade de dados em uma tabela em um sistema de coordenadas em 2D baseado com primeira coordenado sendo linha e segunda a coluna. O HBase™ armazena uma parte dos dados em uma tabela baseada em um sistema de coordenadas 4D. Considerando-se o conjunto completo de coordenadas como uma unidade é possível pensar no HBase™ como um sistema de armazenamento *chave-valor*. Com essa abstração do modelo de dados em mente, pode-se considerar as *coordenadas* como uma *chave* e os *dados da célula* como o *valor*.

A API HBase™ é construída de tal forma que não é preciso fornecer o caminho inteiro de coordenadas ao solicitar dados em uma célula. Se for omitida a versão em seu pedido *Get*, o HBase™ fornece-lhe um mapa de versões para aquele valor. Ao fornecer coordenadas cada vez menos específicas em sua solicitação, o HBase™ permite obter mais dados em uma única operação. Dessa forma, pode-se pensar no HBase™ como um sistema de *chave-valor*, onde o valor é um mapa, ou um mapa de mapas.

4.6.2.2 Particionamento (Sharding)

Com o HBase™ podemos realizar o sharding, que é o particionamento de grandes BDs em bancos menores, distribuídos em servidores diferentes, mais facilmente gerenciáveis, também chamados de data shards. O sharding de BDs é equivalente ao *Particionamento Horizontal.*[6] A unidade básica de *escalabilidade* e *balanceamento de carga* em HBase™ é chamado de *Região*. Como pode ser visto na Figura 20 estas são servidas por servidores de região *RegionServers*, que são processos Java tipicamente alocados com *DataNodes*, (GEORGE, 2011).

As regiões são faixas contíguas essencialmente de linhas armazenadas em conjunto. Eles são divididos de forma dinâmica pelo sistema quando eles se tornam muito grandes. Alternativamente, eles também podem ser fundidas para reduzir o seu número e os arquivos de armazenamento necessários. As regiões HBase™ são equivalentes a partições de intervalo, usado sharding de BDs, elas podem ser espalhados por vários servidores físicos, distribuindo assim a carga e, portanto, fornecendo escalabilidade, (GEORGE, 2011).

[6] O particionamento horizontal divide uma tabela em várias tabelas. Cada tabela contém o mesmo número de colunas, mas menos linhas.

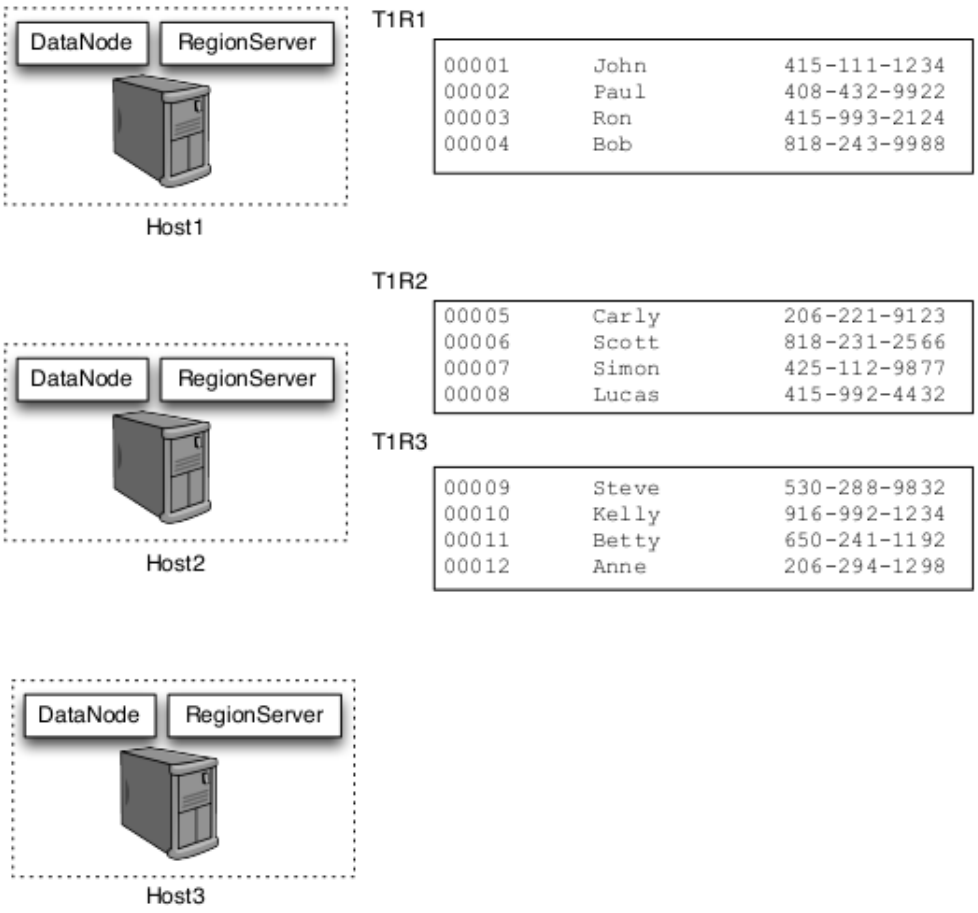

Figura 20 – Particionamento HBase™ da Tabela e mapeamento para os hosts, (GEORGE, 2011).

Cada região pode viver em um nó diferente e é composto de vários arquivos HDFS e blocos, cada um dos quais é replicado pelo Hadoop™.

Inicialmente, há apenas *uma* região para uma tabela, e adicionando dados a ela, o sistema é monitora isto para garantir que não exceda o tamanho máximo configurado. Se ultrapassar o limite, a região é dividida em duas partes com metades aproximadamente iguais de criação de região. Cada região é servida por exatamente um servidor da região, e cada um desses servidores pode servir muitas regiões a qualquer momento. A Figura 21 mostra como a visão lógica de uma tabela é na verdade um conjunto de regiões hospedados por muitos servidores de região, (GEORGE, 2011).

As regiões permitem a recuperação rápida quando um servidor falhar, e balanceamento de carga de *baixa granularidade* (dados mais detalhados), uma vez que podem ser movidos entre servidores quando a carga do servidor que atualmente atende a região está sob muita utilização, se esse servidor ficar indisponível devido a uma falha ou porque está sendo desativada. Particionar a região é um processo muito rápido, perto de instantâneo,

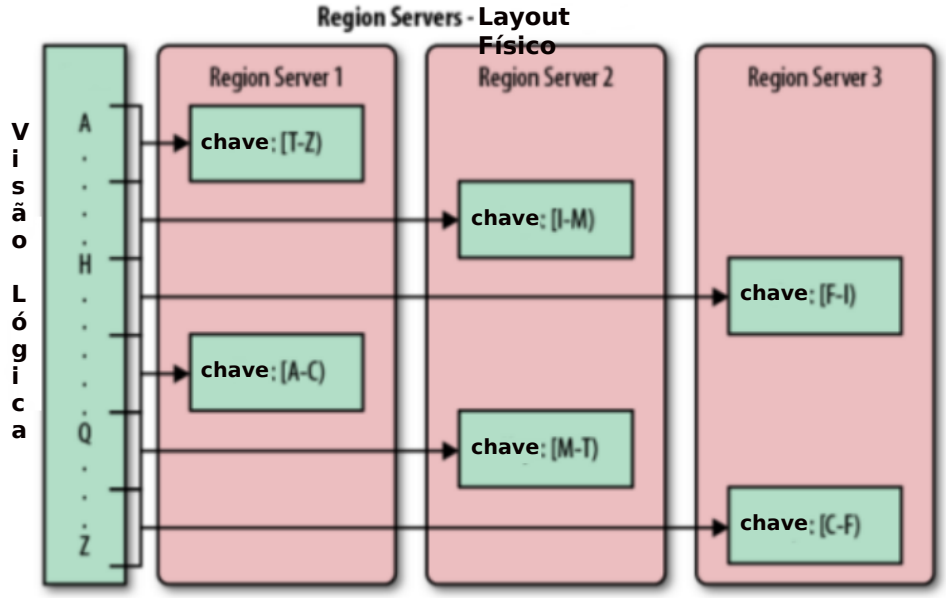

Figura 21 – Visão lógica de uma Tabela HBase™ particionada em regiões, (GEORGE, 2011).

porque as regiões divididas simplesmente lêem a partir dos arquivos de armazenamento originais até a sua compactação reescreveendo em contas separadas de forma assíncrona, (DIMIDUK et al., 2012).

4.6.2.3 Sistema de Replicação

O sistema de replicação do HBase™ possui registros *(HLogs)* dos processos ocorridos, permitindo que se tenha conhecimento integral de cada estágio da replicação e mantendo o servidor sempre informado a respeito de erros e rotinas concluídas.

Os *HLogs* de cada *RegionServer* são a base da replicação do HBase™, e devem ser mantidos no *Hadoop Data File System (HDFS)* por tanto tempo quanto for necessário replicar dados para qualquer cluster escravo. Cada *RegionServer* lê do log mais antigo que ele necessite para replicar e mantém a posição corrente dentro do *ZooKeeper*.

O Zookeeper é um serviço que faz o papel de coordenador e gerenciador de quase todas as principais atividades de replicação, para simplificar a recuperação a falhas. A posição do arquivo pode ser diferente em cada cluster escravo, o mesmo ocorre para consultas de *HLogs* para processar, como mostrado na Figura 22, que ilustra o momento de uma replicação no HBase™.

Assim que o *HRegionServer* efetua a leitura dos logs referentes à replicação, há uma chamada síncrona aos *clusters escravos* para que haja replicação de determinado

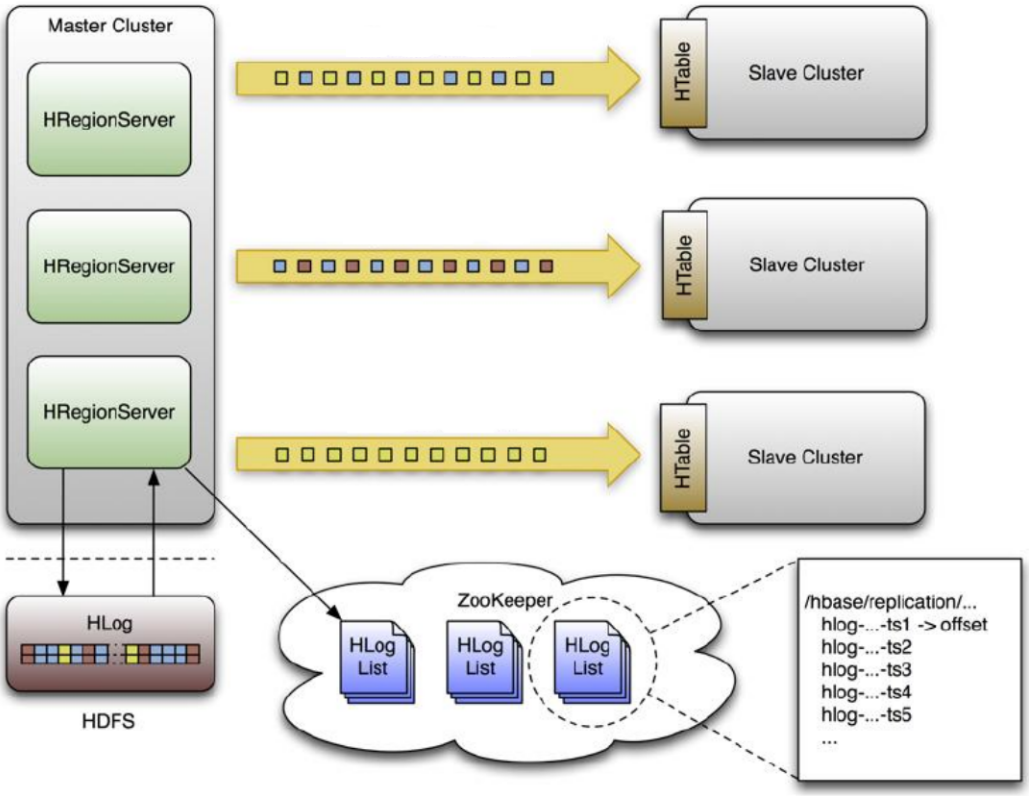

Figura 22 – Esquema HBase^{TM} de Replicação, (GEORGE, 2011).

dado. Os mesmos que participam da replicação podem ser de tamanhos assimétricos e o cluster *mestre* irá fazer o *melhor esforço* para balancear o fluxo de replicações entre os *escravos* confiando na randomização.

4.7 BANCO DE DADOS TEMPORAL (BDT)

Bancos de dados temporais armazenam informações que variam de acordo com o tempo. Ou seja, criam uma espécie de histórico com o qual várias análises podem ser realizadas, trazendo uma maior compreensão do fenômeno estudado, como por exemplo tráfego em redes de dados, (MITSA, 2010). Apesar de ser uma área estudada a muito tempo, ainda oferece um grande número de desafios.

Segundo **??**), um dado pode ser caracterizado como temporal quando apresenta mais de um estado ao longo do tempo. Um estado pode ser compreendido como um valor associado a um objeto durante um certo intervalo de tempo. Um objeto tem seu estado alterado por meio um evento. Um evento pode ser visto como um fenômeno instantâneo. Tais definições nos levam a uma correlação entre estados e eventos. Em uma análise mais simples, um objeto temporal pode ser visto como um conjunto de atributos que podem

assumir diferentes valores em intervalos de tempo distintos. Em um determinado instante de tempo, porém, eles possuem apenas um valor. Esta definição nos leva diretamente ao conceito de *sequência de dados (timesequence)*.

Uma vez compreendido o que é um dado temporal, é necessário analisar o que representa uma *marca de tempo (timestamp)*. A marca de tempo é utilizada para indicar quando um atributo assume um novo valor. Nesta analise, serão considerados dois principais tipos de marcas de tempo: o *tempo de transação* e o *tempo válido*. O *tempo de transação* representa o instante no tempo em que um certo registro foi adicionado no *SGBD*. O *tempo válido*, por sua vez, representa o tempo em que um fato realmente ocorreu no mundo real.

De acordo com Mitsa (2010), Os bancos de dados que suportam timestamp podem ser dividido em quatro categorias:

- *Bancos de dados Snapshot:* Mantêm-se a versão mais recente dos dados. Bases de dados convencionais se enquadram nesta categoria.
- *Bancos de dados Rollback:* Eles suportam apenas o conceito de tempo de transação.
- *Bancos de dados Históricos:* Eles suportam apenas de tempo válido.
- *Bancos de dados Temporais:* Eles utilizam ambos os timestamps: *válidos* e de *transação*.

Ainda segundo Mitsa (2010) as unidade de tempo para BDTs são:

- *Intervalo:* A entidade temporal com uma hora de início e uma hora de término.
- *Evento:* A entidade temporal, com um tempo de ocorrência.

Uma parte interessante a destacar relativa aos *timestamps* na sua utilização no HBase™, é que como este utiliza células com *versões*, portanto numa situação onde se quer obter a versões (uma lista) de *timestamps* para um determinado *alerta* ou *evento* de recurso da Nuvem, ou seja, fazer uma análise de eventos para aquele recurso conforme uma *linha de tempo*, o que é muito útil no caso de Monitoramento. Isso é possível por intermédio da sua *API* em *Java*™, mais diretamente criando-se um Cliente neste linguagem e utilização da Classe *TimestampsFilter* e o método que retorna uma lista de *timestamps List<Long> getTimestamps()*, essa utilização será vista no experimento do capítulo seguinte.

4.7.1 Satisfação das Restrições Temporais (SRT) e Padrões Intervalares

Para (SCHWALB; VILA, 1998), a *satisfação da restrição temporal (SRT)* é uma *TI* útil para representar e responder consultas sobre ocorrências temporais e as relações temporais entre eles. A informação é representado como um *problema da satisfação de restrições (CSP)*, onde *variáveis* denotam os *horários dos eventos* e as *restrições* representam as *possíveis relações temporais entre eles*.

Em Mitsa (2010), é analisado as *restrições temporais (relações temporais entre horários de eventos)* e sua classificação. podem ser *qualitativas* ou *quantitativas*.

- Quanto restrições temporais *quantitativas*, as *variáveis (horários dos eventos)* têm seus valores sobre o conjunto de entidades temporais e as *restrições (relações destes horários)* são impostas a uma variável, restringindo o seu conjunto de valores possíveis.
- Em restrições temporais *qualitativas*, as variáveis (horários dos eventos) tiram o seu valor a partir de um *conjunto de relações temporais*.

Em 1983, Allen desenvolveu a Lógica Temporal de Intervalos (LTI), posteriormente aprimorada em, que é signicantemente mais expressiva e mais natural para representar o tempo que outras aproximações, como por exemplo em Inteligência Articial. Na LTI, ele considera um período de tempo como sendo o tempo associado a algum evento e propõe representações e considerações para descrever a estrutura temporal básica usada na lógica.

Para relacionar dois intervalos ele definiu *N* relacionamentos possíveis, através de predicados temporais. Por exemplo, *before(i,j)* significa que o intervalo i ocorre antes do intervalo j e *after(j,i)* é o relacionamento inverso, *meets(i,j)* signica que o intervalo i termina o mesmo instante em que o intervalo j começa e *metby(j,i)* é o relacionamento inverso, e assim por diante. O conceito de *relações temporais*, é definido desta forma indicando um determinado intervalo. Portanto teremos relações formalmente definidas como: *bi(X, Y) = b(Y, X)*. Na Figura 23 segue exemplo de outras relações mostradas em Mitsa (2010).

Uma expansão dessa teoria é baseada em *intervalo de tempo* da relação anterior e adicionado *pontos* como entidades de interesse. Existem *dois tipos de entidades de ponto*: *pontos* e *momentos*, definidos por *Allen*. Os pontos são definidos como *pontos* de encontro dos períodos, enquanto *momentos* são, períodos muito pequeno de tempo não-degradáveis. Para Allen, períodos de tempo são intervalos e os limites que definem o início e o fim dos

Nome	Notação
Before	$b(X,Y)$
Overlaps	$o(X,Y)$
During	$d(X,Y)$
Meets	$m(X,Y)$
Starts	$s(X,Y)$
Finishes	$f(X,Y)$
Equals	$e(X,Y)$
After	$bi(X,Y)$
Overlapped-by	$oi(X,Y)$
Contains	$di(X,Y)$
Met by	$mi(X,Y)$
Started by	$si(X,Y)$
Finished by	$fi(X,Y)$

Figura 23 – Relações Temporais, (MITSA, 2010).

intervalos são considerados pontos, por isso, ele não faz relacionamentos entre pontos (não períodos) e intervalos (períodos).

Padrões temporais que representam o tempo por intervalos são aqueles em que os eventos ocorrem em determinados períodos de tempo durante o período entre duas horas, durante o período entre duas datas, etc. Estes padrões são chamados de *padrões intervalares*. Grande parte dos autores de trabalhos sobre mineração de padrões intervalares se baseiam em um formalismo lógico, chamado de *lógica temporal de intervalos (LTI)*, desenvolvido por *Allen*.

Estes conceitos são úteis na primeira parte da fase de *Análise* do ciclo MAPE-K da arquitetura de Schubert (2014), onde é feita a analise dos dados contidos no *banco de monitoramento* (alvo de modelagem neste trabalho), clusterização e subsequente classificação dos mesmos. Um exemplo típico para Nuvem onde as *restrições temporais* são determinantes, é o Cloud Balancing, onde existem vários processos para ser distribuidos entre um certo número de servidores da Nuvem, cada qual com sua capacidade de CPU, RAM, etc. E, é necessário atribuir os processos de forma a obter o menor custo de manutenção do serviço.

Enquadrando estes conceitos na utilização de Banco de Dados Temporal-Intervalar para o Armazenamento das Informações de Monitoramento da Nuvem, abrem-se novas possibilidades de análise e mineração dos dados. Isto permite que Algoritmos de Descoberta de Conhecimento valham-se das *relações temporais* para descobrir novos padrões

de comportamento do sistema, bem como relações de causalidade que não poderiam ser estabelecidas se por um registro atômico da ocorrência de eventos, sem a sua respectiva duração.

4.7.2 Série Temporal (time-series)

Outro conceito vinculado a sequências de dados são as *séries temporais*. Uma série temporal, assim como uma sequência de dados, representa uma sequência de valores associados a um objeto, sendo que o objeto não possui dois valores distintos no mesmo instante de tempo, (RAFIEI; MENDELZON, 1997). As principais diferenças entre *séries temporais* e *sequência de dados* são relativas ao volume de dados e ao conjunto de operações necessário para realizar todas as manipulações desejadas com os dados brutos.

Os dados de *séries temporais* são como uma coleção de *pontos de dados (PD)* ou *tuplas*. Cada ponto tem um *timestamp* e uma medição. Este conjunto de pontos ordenados pelo tempo é a *série temporal*. As medições são normalmente coletados em um intervalo regular, o que é muito utilizado no caso de *monitoramento de redes (MR)*, demonstrado na Figura 24.

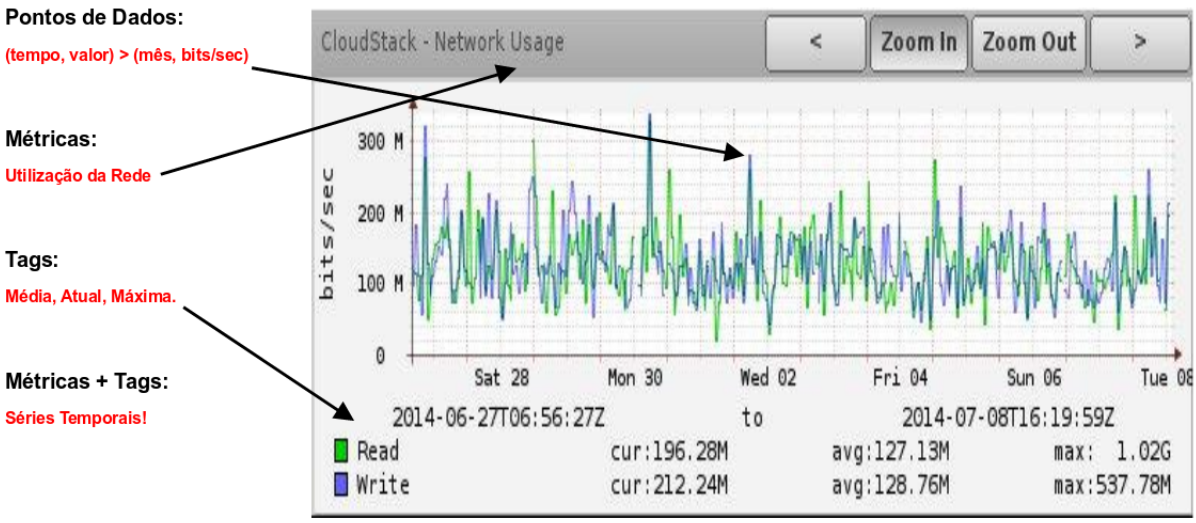

Figura 24 – Ponto de dados em uma métrica de Monitoramento do CloudStack™, autoria nossa.

É comum que esses pontos de dados também carreguem *metadados* sobre a medida, como o nome do *"host"* gerando a série. Ao ligar uma *timestamp* à uma medição, podemos entender as diferenças entre os valores de medição na progressão do tempo e também entender os padrões ao longo do tempo. Por exemplo, a temperatura da CPU de um

determinado servidor pode ser medido a cada hora. É natural supor pontos anteriores pode informar um ponto futuro. Portanto, é possível adivinhar a temperatura próxima de hora com base em medições dos últimos cinco horas. Este é o conceito elementar aplicado na *tomada de decisão(TD)* para a arquitetura de monitoramento em Nuvem proposta, podendo prever outros dados, é possível tomar decisões inteligentes em relações as métricas da Nuvem fazendo inferencias com PDs intervalares.

Dados em *séries temporais* podem ser um desafio a partir de uma perspectiva de gerenciamento de dados. Todos os *pontos de dados (PD)* em um sistema podem compartilhar os mesmos campos, por exemplo: *data/hora*, *localização* e *medição*. Mas dois PDs com valores diferentes para qualquer um destes campos podem ser completamente independentes. Se um ponto é a temperatura na CPU do Servidor Cirrus do LRG e outro do Stratus, eles não estão provavelmente relacionados, até mesmo com um *"timestamp"* similar, (DIMIDUK et al., 2012). Este conceito é essencial no nosso trabalho para determinar a row-key do nosso esquema de *BD* HBase™, por que como veremos no próximo Capítulo, na seção do modelo de dados, a escolha da row-key é determinante para alguns problemas que podem surgir.

Outra questão notável com séries temporais é a gravação de dados. As *árvores* são uma estrutura de dados eficiente para acesso aleatório, mas um cuidado especial deve ser tomado quando construí-las em ordem de classificação. Uma série temporal é naturalmente ordenada por tempo e é muitas vezes persistiu de acordo com essa ordem. Isto pode resultar em estruturas de armazenamento a ser construídas da pior forma possível, tal como ilustrado Figura 25.

Assim como uma árvore, esta ordem também pode causar estragos em um *sistema distribuído (SD)* e o Apache HBase™ é distribuído em *árvore B*[7]. Quando os dados são particionados entre os nós da árvore de acordo com a data e hora, surgem novos gargalos de dados em um único nó, causando um *hotspot*. Como o número de clientes do *SD* que escrevem dados aumenta, um único nó é facilmente dominado gerando falhas.

[7] Estrutura de dados muito empregada em aplicações que necessitam manipular grandes quantidades de informação tais como um banco de dados ou um sistema de arquivos.

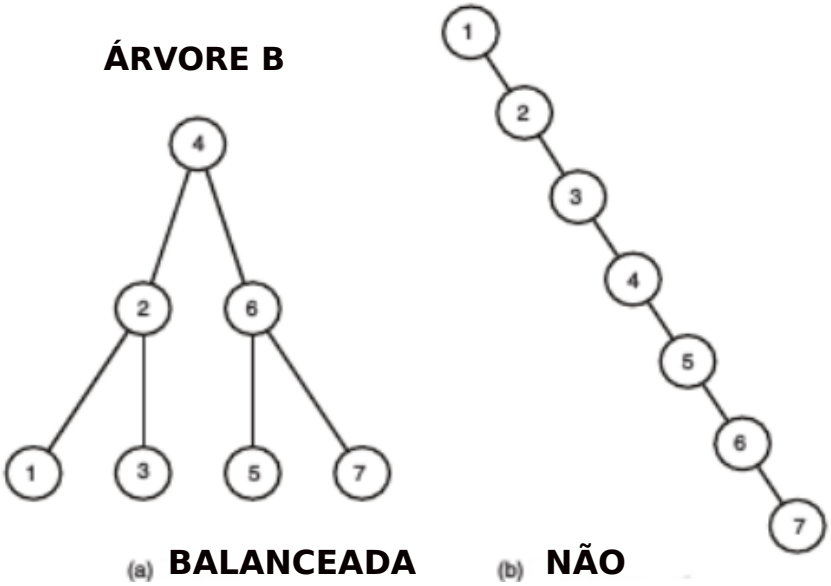

Figura 25 – Árvore B em time-series balanceada e não balanceada, (DIMIDUK et al., 2012).

4.7.3 Séries Temporais no Monitoramento em Nuvem

De acordo com Dimiduk et al. (2012), o Monitoramento de uma infraestrutura seja ela uma Rede ou Nuvem é a definição utilizada quando se quer manter o controle sobre os sistemas implantados. A grande maioria dos projetos de software são implementados como sistemas e serviços online de comunicação através de uma rede. Questões relevantes no *MN* passaram por saber se o sistema está *ligado* ou *desligado*, se houve o controle de quantas solicitações serviu a cada hora ou que horas do dia, se o sistema sustentou a maior parte do tráfego, se disparou algum alerta no meio da noite por causa de um serviço que caiu, etc.No entanto, *MN* é muito mais do que a notificações e alertas. A série de eventos que provocou o alarme a qualquer hora que seja representam apenas uma pequena quantidade do total de dados dessas ferramentas à coletar.

Os *pontos de dados (PD)* relevantes incluem solicitações de serviço por segundo, sessões de usuários ativos simultâneos, escrita e leitura na BD, a latência média de resposta, o consumo de memória dos processos, e assim por diante. Cada um deles é uma medida de *séries temporais (ST)* associados a uma métrica específica e individualmente fornece apenas um pequeno *snapshot (instantâneo)*[8], de visibilidade do funcionamento global do sistema. Tomando estas medidas em conjunto ao longo de uma janela de tempo comum, e tem-se uma visão prática do sistema em execução.

[8] Em sistemas de computador, um instantâneo é o estado de um sistema em um determinado ponto no tempo.

4.8 HBASE™ NO CONTEXTO TEMPORAL-INTERVALAR

O HBase™ é uma excelente escolha para aplicações que utilizam séries temporais, devido ao fato de fornecer armazenamento de dados de forma *escalável* - quando se pode atualizá-lo, para aumentar o processamento de transações, mediante a adição de novos processadores, mecanismos e dispositivos de estocagem, que podem ser atualizados facilmente e de modo transparente, sem precisar desligar o sistema, com suporte para consultas de *baixa latência*[9], (DIMIDUK et al., 2012), conceito este explorado no experimento que vai ser analisado no próximo capítulo.

É um armazenamento de dados *propositalmente flexível*, que permite aplicações que utilizam *séries temporais* elaborarem um esquema eficiente e relativamente personalizado para armazenar os seus dados.

No modelo de dados proposto no próximo capítulo, este é personalizado para medições de séries temporais e suas *"tags"* (etiquetas) associadas. O que vai ao encontro das necessidade de adaptação do modelo MySQL® innoDB. Isso porque a grande maioria dos *atributos* das tabelas convencionais de *BDRs* podem ser tratados como *"tags"*.

O HBase™ fornece *consistência* forte, como pode ser visto na seção de teorema CAP, então a visualização no *"dashboard"* das séries temporais de cada métrica analisada na Nuvem pode, também, ser usada para geração de relatórios em tempo real. A exibição dos dados coletados no HBase™ fornece é sempre o mais atualizado possível devido ao fato do CloudStack™ aderir a propriedade de Pontualidade de Aceto et al. (2013), e o processo de leitura em massa de dados por parte deste banco de dados é atômico. A *escalabilidade horizontal* dele é crítica por causa do volume de dados necessário para o esquema apresentado. Certamente, outros bancos de dados podem ser considerados, (DIMIDUK et al., 2012). Pode-se fazer um *"backup"* no *MySQL*®. por exemplo, mas a implantação depois de um mês de coleta de centenas de milhões de pontos de dados (PD) por dia, meses, anos em um *ambiente de nuvem (AN)* real com diversos aplicativos heterogêneos em execução torna inviável a extensão da máquina MySQL®, original para lidar com o volume de dados.

A questão se resume ao *custo* e *complexidade*. O *dimensionamento* de um

[9] Período de latência é a diferença de tempo entre o início de um evento e o momento em que seus efeitos se tornam perceptíveis e sim conceito de engenharia, o débito.

sistema relacional para lidar com esse volume de dados requer uma estratégia de particionamento. Tal abordagem, muitas vezes coloca o ônus de particionamento no código do aplicativo. Seu aplicativo não pode requisitar poucos dados do banco de dados. Ao invés disso, ele é forçado a encontrar qual o banco de dados hospeda os dados em questão com base em seu conhecimento atual de todos os *"hosts"* disponíveis e todos os intervalos de dados, (DIMIDUK et al., 2012).

Além disso, ao particionar os dados, perde-se a principal vantagem de sistemas relacionais: a *linguagem de consulta poderosa*. Os dados particionados são espalhado por vários sistemas que desconhecem uns aos outros, o que significam que as consultas são reduzidos a pesquisas de valores simples.

Qualquer complexidade adicional à consulta é refletida no código do cliente de consulta. Já o HBase™ esconde os detalhes de particionamento dos aplicativos clientes. As partições são alocadas e gerenciadas pelos *clusters* para qual o código do aplicativo continua a desconhecer. Isso significa menos complexidade para gerenciar o código cliente de consultas. Embora o HBase™ não suporte uma linguagem de consulta rica como o SQL, é possível projetar o esquema HBase™ de tal forma que a maior parte da complexidade de consulta on-line está praticamente toda nosx+ *clusters*. Os coprocessadores do HBase™ dão-lhe a liberdade para inserir os dados em linhas arbitrária nos nós que hospedam os dados. Além disso, ele também têm o poder de MapReduce™ para consultas off-line, dando-lhe uma grande variedade de ferramentas para a construção de de relatórios e gráficos, (DIMIDUK et al., 2012).

5 MODELO DE DADOS, ESTUDO DE CASO EXPERIMENTO

Este trabalho formaliza uma série de conceitos teóricos, sua análise de maneira top-down e devidas correlações com o intuito de justificar o modelo de armazenamento de dados temporal-intervalar. Foram apresentados os conceitos que justificam uma arquitetura cognitivo autônoma para monitoramento em nuvem (ACAMN), conceitos relacionados à computação em nuvem, monitoramento, suas tecnologias, ferramentas e a *taxonomia de monitoramento* proposta por Aceto et al. (2013), onde são elencadas propriedades importantes para verificar se uma ferramenta de monitoramento adere ou não ao paradigma em Nuvem. Com isto foi possível verificar que a maioria das ferramentas de código aberto atuais atendem no máximo *duas ou três* das propriedades de Aceto et al. (2013). Além das propriedades da taxonomia analisada foi possível verificar que apenas uma das ferramentas de código aberto atuais utiliza BDs NoSQL, o framework Sensu™.

Com a contextualização de *big data* e *banco de dados temporais intervalares* abrem-se assim novas possibilidades de análise e mineração dos dados. Os algoritmos de descoberta de conhecimento utilizam essas relações para descobrir novos padrões de comportamento do sistema, bem como relações de causalidade que não poderiam ser estabelecidas só por um registro atômico da ocorrência de eventos, sem a sua respectiva duração. Escolhidos pelo LRG, com propósitos de atividades de pesquisa, o CloudStack™ e o Zenoss Core® com o seu plug-in Zenpack Zenoss Inc. são as principais ferramentas utilizadas neste trabalho e possuem como principal propriedade da *taxonomia* a *Pontualidade* - Eventos detectáveis estão disponíveis a tempo da utilização proposta, evitando latência para a criação de *snapshots* da Nuvem - o que se torna interessante para a solução de modelagem de dados de informações de uma Nuvem, pois o *modelo* proposto é para um BDT Intervalar. O fato das ferramentas em utilização terem essa característica vai ao encontro das possíveis pesquisas a serem realizadas em *tomada de decisão* e também implementação da arquitetura de monitoramento referida.

Relativamente as métricas de monitoramento de uma Nuvem, que são os dados coletados e armazenados na *BD de monitoramento* vista na arquitetura ACAMN. Estes dados são úteis para as demais etapas da arquitetura. E, na parte onde os dados são disponibilizados para o ciclo MAPE-K, ou seja, quando se inicia a análise, com as relações

temporais corretas é possível filtrar, clusterizar e armazenar as informações corretas para Nuvem que levam a mesma à uma qualidade de serviço, eficiência energética e segurança se fizer a sua auto-gestão de forma eficiente com análise das informações monitoradas.

No capítulo anterior também foram apresentadas as tecnologias de *SGBDs* SQL aderentes as propriedades *ACID* e NoSQL aderentes ao teorema CAP, mas na última seção foi analisado o pormenor do porquê de se utilizar o HBase™ no modelo de dados intervalar-temporal da seção a seguir.

5.1 MODELO DE DADOS

.

O *modelo* proposto é um aperfeiçoamento da implementação atual do banco de dados *MySQL innoDB* utilizados atualmente. O armazenamento e consulta de dados em *séries temporais* eficiente é algo para o qual o modelo relacional padrão não é adequado como pode ser visto anteriormente. O modelo proposto não vai causar a perda de informação, pois é flexível o suficiente a ponto de armazenar a mesma informação que tabelas distintas armazenavam sem perda de dados, pois toda a informação armazenada por essas tabelas podem de alguma forma ser acomodadas no modelo proposto. Isto é um ganho muito evidente pois de duas tabelas (ou mais tabelas) que teriam que ser mineradas separadamente passamos os dados todos para um única tabela onde as ferramentas de mineração de dados podem atuar de forma mais eficiente, ou podem ser feitas consultas interessantes para tomada de decisão.

De acordo com Kai et al. (2013), os valores de métricas precisam ser armazenados persistentemente para a *Análise* do ciclo MAPE-K, bem como exibidas atualizadas de acordo com a propriedade de Pontualidade de Aceto et al. (2013), esta ultima inferência é feita pelo autor deste trabalho. Os recursos na Nuvem alteram-se dinamicamente e a implantação da Nuvem pode ser grande. Então, monitorar este sistema distribuído pode produzir uma grande quantidade de valores de métricas. Assim, o sistema de armazenamento deve ser *escalável* e *flexível*, com a capacidade de coletar milhares de métricas a partir de milhares de hosts e aplicativos a um ritmo elevado. É esse o ambiente que se espera monitorar e que justifica um contexto big data para monitoramento de Nuvens, e com o banco de dados HBase™ será possível realizar transações de forma escalável para as analise que se propõe.

Ainda segundo Kai et al. (2013), uma vez que a maioria dos valores de métrica tem as propriedades de *séries temporais* (vários valores por objeto (métrica) numa dada *timestamp*, várias operações para tratar dados brutos e volume de dados considerável), foi escolhido o BD de séries temporais intervalar. Os fornecedores de bancos de dados relacionais, muitas vezes olham para soluções fora do padrão para este problema, como armazenar os dados de séries temporais em blobs ou fornecendo extensões proprietárias de consulta para sua introspecção, (DIMIDUK et al., 2012). Os dados de *séries temporais* tem características distintas. Estas características podem ser exploradas por estruturas de dados personalizadas para armazenamento e consultas mais eficientes. Mas, os sistemas relacionais não suportam nativamente esses tipos de formatos de armazenamento especializados, de modo que essas estruturas são muitas vezes serializadas em uma representação binária e armazenados como uma matriz de bytes não indexados. As operadores personalizados são então obrigados a inspecionar esses dados binários. Os dados armazenados em um pacote como este é comumente chamado de blob, (DIMIDUK et al., 2012).

Em comparação com dados em BDRs, os banco de dados de séries temporais com intervalos podem lidar com milhões de séries temporais com acréscimo mínimo no tempo de extração de dados e com os requisitos de espaço em disco limitado. Isto é importante para a escalabilidade a utilização do Apache HBase™ como já foi descrito, que pode ser facilmente escalável para armazenar milhares de valores de métricas. Uma observação que deve ser feita, é que a estrutura de armazenamento proposta neste trabalho pressupõe um *coletor de dados* adaptado às ferramentas com intuito de fornecer os dados já mapeados, compactados e representados da armazenados da maneira correta e como se sugere no modelo de dados, (CHIFU et al., 2009; KAI et al., 2013; PRASAD; AVINASH, 2013).

Ademais, é pensado na utilização de conceitos de computação autônomica e cognitiva relacionados à arquitetura de monitoramento proposta e um ambiente de Nuvem em larga escala que gera grande massa de dados, o que implica em tomar decisões importantes durantes as consultas de relações temporais e de forma muito controlada, a atenção a isto é necessária para a alocação de recursos consoante as necessidades da Nuvem consultando seu último estado registrado. Por fim, no ambiente em Nuvem heterogêneo e em larga escala como sugerido, a arquitetura de monitoramento beneficia o auto-gerenciamento,

auto-configuração, auto-cura, auto-otimização e auto-proteção, justificando a utilização de um modelo de dados temporal-intervalar com um BD NoSQL e garantindo assim a qualidade de serviço (QoS), eficiência energética e segurança da Nuvem.

5.1.1 Estudo de Caso

Um esquema de bancos de dados de um SGBD é sua estrutura descrita em uma linguagem formal suportada por este e refere-se à organização de dados como um diagrama de como um banco de dados é construído (dividido em tabelas de banco de dados no caso de bancos de dados relacionais). Um dos objetivos deste trabalho é investigar as possíveis opções deste esquema.

As duas situações que mais se enquadram estão discutidas em listas, HBASEUSERLIST (2010), apresentações, George (2011), Khurana (2013) ou como recomendação na documentação, HBASEGUIDE (2014) do HBase™. Existem diversos livros sobre o HBase, por exemplo, George (2011), Dimiduk et al. (2012), tratando da modelagem de chaves ou da migração de um esquema relacional para o NoSQL, mas nenhum trata tão a fundo a questão da modelagem para monitoramenteo em time-series como o OpenTSDB™[1], (OPENTSDB, 2014).

A melhor escolha da *chave de linha* e *esquema* para o HBase™ é uma das mais importantes tarefas para definir um bom modelo. O primeiro problema a se pensar antes de definir a *chave de linha* é o caso de métricas onde muitas informações são necessárias a serem armazenadas apenas em uma linha, o qual é chamado de “overhead” da linha, Khurana (2013), George (2011). Portanto para adequar a realidade de várias tabelas de um esquema MySQL® à do HBase™ é necessário realizar a denormalização de todas as tabelas do MySQL® para no fim não termos uma linha extensa de dados em array de bytes onde o limite por linha é de apenas 50MB, Dimiduk et al. (2012) e uma consulta com tantas informações por linha é extremamente ineficiente, (GEORGE, 2011). Para contornar o problema, é preciso adaptar toda denomarlizaçao do MySQL® criando no HBase™ *duas tabelas*, onde todos atributos das tabelas denormalizadas do MySQL® são tratados como “tags” (etiquetas), e uma tabela referência a outra por uma “chave estrangeira” onde na segunda tabela são armazenadas “tags” e seu versionamento com um “timestamp” atômico.

[1] É um daemon de séries temporais e com suporte a linha de comando, onde é possivel obter métricas de monitoramento de rede das mais variadas topologias de rede e dispositivos.

Esta solução podera se torna mais visível com um exemplo do mapeamento da tabela “alert” do CloudStack™ que é mostrado logo após a definição da chave de linha do esquema de dados. Lembrando que para o Apache HBase™ não existe chave estrangeira, é apenas uma referência para contextualizar com o paradigma relacional.

Partindo da bibliografia consultada e já citada, temos a duas opções: a primeira é problemática e pior opção; a segunda opção é complexa mas juntando todos os conceitos expostos, principalmente de pontos de dados em séries temporais e BDT intervalar, é possível verificar que este modelo com o esquema adotado é eficiente e já praticado de outra maneira pela ferramenta OpenTSDB, (OPENTSDB, 2014), que inclusive adota o mesmo SGBD. As possibilidades de escolha (estudo de caso) de *chaves de linha* para o *esquema* são as duas (2) que seguem:

- Quando uma *chave de linha* é apresentada como na Tabela 1 e, como pode ser observado nas recomendações da documentação HBase™, (HBASEGUIDE, 2014): Existe um problema inerente de *crescimento monotônico*[2] pois a *timestamp* é incremental crescendo ao longo do tempo. Portanto, qualquer processo envolvendo uma *Scan* ou *Get* de dados pode entrar em *lock-step* - situação em que a execução fica bloqueada pois há duas transações com a mesma característica (Scan ou Get) a concorrerem para qual irá ser executado primeiro, pois atingem a mesma *HFile* muito reduzida (apenas uma linha, por exemplo) após muitos particionamentos (da polítca de autosharng horizontal do HBase e do “timestamp” estar na primeira posição da “row-key”, lembrando que o timestamp têm crescimento monotônico e isso faz ele sempre criar uma nova linha e as HFile se dividem até se tornarem do tamanho de uma linha) - porque existindo vários *clientes* criando processos de importação (Scans e Gets), estes acabam por atingir a mesma *Região (HRegion)* da tabela (e, portanto, um *único nó*) do *HBaseServer*, consequentemente uma única linha do HFile, o que discarta a hipótese de utilização desta *row-key (chave de linha)*, uma vez que estas transações são muito importantes para uma ferramenta de análise intervalar das métricas em série temporal, ou seja, as leituras de dados.

- O formato de chave recomendado na literatura, HBASEGUIDE (2014), OPENTSDB

[2] Diz-se das funções que crescem, decrescem, ou são constantes em uma dada linha de tempo, neste caso em específico é o crescimento, mono + tom, único tom.

Tabela 1 – Chave de linha com crescimento monotônico e lock-step.

*{*Timestamps*}* Origem

(2014) é o hoje utilizado pelo OpenTSDB™. É utilizada uma *"row-key"* com o formato mostrado na Tabela 2, o que parece à primeira vista parece contradizer o conselho anterior sobre não usar um *timestamp* como a chave devido ao *crescimento monotônico*. No entanto, a diferença é que o *tempo* não está na *posição principal* da chave, e é pressuposto por concepção a existência de dezenas ou centenas (ou mais) dos diferentes tipos de origem. Assim, mesmo com um fluxo contínuo de dados de entrada, e com uma mistura de tipos de origens, os *Puts* são distribuídos através de vários pontos de *Regiões (HRegion)* na tabela e os *Scans* ou *Gets* não sofrerão *lock-step* pois não há muito "autosharding" (particionamento horizontal) do HBase™.

Tabela 2 – Chave de linha no estilo do OpenTSDB™ adotada.

Origem *{*Timestamps*}{*Tag*}*

5.2 ESQUEMA

Na próxima Tabela 3, é motrada o *Formalismo de Extended Backus-Naur (EBNF)*[3] para o esquema teórico proposto. O armazenamento do *banco de monitoramento (BM)* da *ACAMN* é logicamente feito com retroalimentação de dados em uma *série temporal* de *timestamps*, o que significa ter uma *origem (UID)* de valor fixo, o *dado* relacionado a ela também, mas as suas "tags" e timestamps finais e inicias se alteram, assim que alguns valores dos atributos (no modelo relacional, mas tags neste modelo) com "timestamp" atômico também podem se alterar, por serem por exemplo, uma "flag".

Por fim, como se pode ver no *esquema teórico* descrito e mostrado na Tabela 3, existem campos *tags* (etiquetas) - uma palavra-chave (relevante) ou termo associado com uma informação (ex: tipo de alerta, qual dispositivo, mensagem de alerta) que o descreve e permite uma classificação da informação baseada em palavras-chave - que representam melhor os *atributos* do esquema relacional em *MySQL*® e Tabelas *innoDB*.

[3] É uma meta-sintaxe usada para expressar gramáticas livres de contexto, isto é, um modo formal de descrever linguagens formais.

Tabela 3 – BNF do esquema de dados temporal-intervalar, autoria nossa.

(i) Esquema = Timestamp Inicial Timestamp Final Origem Data *{*Tag*}*

(ii) Tag = Chave Valor *{*Timestamp Atômico*}*

Para os propósitos do ambiente de Nuvem proposto neste trabalho, com os conceitos relacionados e características já descritos este modelo de dados com o esquema descrito nesta seção não gera restrições pois não vai causar a perda de informação inclusive em um contexto de Big Data, é flexível o suficiente a ponto de armazenar a mesma informação que tabelas distintas em um esquema MySQL® innoDB armazenavam sem perda de dados, pois toda a informação armazenada nas tabelas das ferramentas utilizadas no LRG podem, de alguma forma ser acomodada no modelo proposto. A título de exemplificação é possível ver na Figura 26 de forma mais prática o mapeamento da tabela *alert* do CloudStack®, e a confirmação que modelo proposto acomoda toda informação.

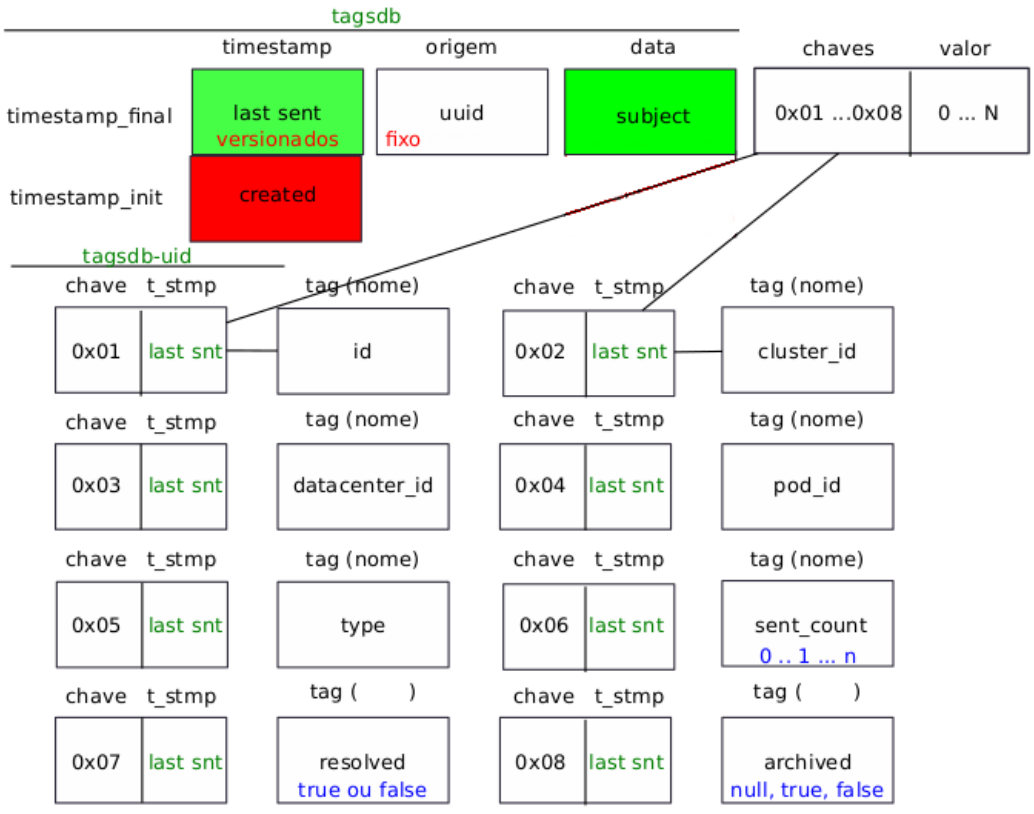

Figura 26 – Mapeamento da Tabela “alert” do CloudStack® no Esquema proposto, elaborada pelo autor.

5.2.1 Tabelas TAGSDB-UID e TAGSDB no HBase™

Começamos esta seção mostrando uma maneira de se criar as duas tabelas estipuladas para a implementação da nossa solução no Apache HBase™. A primeira coisa a notar do Código 5.1 a seguir é como o roteiro é semelhante para qualquer *script* que contém código *data definition language (DDL)* para um banco de dados relacional. O termo *DDL* é freqüentemente usado para distinguir código que fornece definição de esquema e modificação do código de execução de atualizações de dados. Um banco de dados relacional *SQL* usa DDL para modificações de esquema, mas o Apache HBase™ depende da *API*. A maneira mais conveniente de acessar a API para esta finalidade é pelo *HBase Shell*. Outro aspecto a se notar é que foi utilizado a *compressão LZO*, que no arquivo de armazemento *HFile* haverá uma compactação dos *bytes* durante a *escrita* e, portanto, deverá ser descompactado durante a *leitura*. A compressão têm mais vantagens do que desvantagens, o Apache HBase™ só suporta nativamente *GZip* e a habilitação do *ZLO* é um tanto problemática, (HBASECOMPRESSION, 2014), mas para obter a *redução* de *bytes* escritos/lidos no *HDFS*, melhoramento da *largura de banda* e *espaço em disco* e *diminuir* o tamanho dos dados a serem lidos, é uma opção que afeta substancialmente o desempenho com um grande volume de dados armazenados.

Código 5.1 – Shell Script para setup das tabelas do ACAMN - autoria nossa

```
#!/bin/sh # Small script to setup the hbase table used by ACAMN.
test -n $HBASE_HOME || {
  echo >&2 'The environment variable HBASE_HOME must be set'
  exit 1
}
test -d $HBASE_HOME || {
  echo >&2 No such directory: HBASE_HOME=$HBASE_HOME
  exit 1
}
TAGSDB_TABLE=${TAGSDB_TABLE-'tagsdb'}
UID_TABLE=${UID_TABLE-'tagsdb-uid'}
COMPRESSION=${COMPRESSION-'LZO'}
exec $HBASE_HOME/bin/hbase shell <<EOF
```

```
create '$UID_TABLE',
  {NAME => 'id', COMPRESSION => '$COMPRESSION'},
  {NAME => 'name', COMPRESSION => '$COMPRESSION'}
create '$TAGSDB_TABLE',
  {NAME => 'm', COMPRESSION => '$COMPRESSION'}
EOF
```

5.2.1.1 Tabela TAGSDB-UID

O *esquema* utilizado depende do Apache HBase™ por duas funções distintas. A Tabela TAGSDB fornece suporte de armazenamento e consulta sobre dados de séries temporais. A Tabela TAGSDB-UID mantém um índice de valores globais exclusivos para uso como Tags das origens. No Código 5.2, a Tabela TAGSDB-UID contém duas *famílias* de colunas: *ID* e *Nome*. A Tabela TAGSDB também especifica uma *coluna família*, com o nome M (apenas por que é obrigada a especificar uma, mas o nome não têm utilização, como no caso da Nome e ID. Os comprimentos dos nomes das *coluna família* são todos muito curtos. Isso é por causa de um detalhe de implementação do formato de armazenamento *HFile* da versão atual de nomes HBase™ mais curtos significam menos dados para armazenar por exemplo o *valor da chave*. É possível notar a falta de uma abstração de nível mais elevado, diferentemente da maioria dos bancos de dados relacionais populares, não existe o conceito de grupos de Tabela. Todos os nomes de tabela em HBase™ coexistem em um espaço comum gerido pelo *HBase Master*.

Embora a Tabela TAGSDB-UID seja acessória à Tabela de TAGSDB, é explorada sua definição, porque em primeiro lugar a compreensão da sua existência irá fornecer uma visão sobre a concepção global. O projeto do esquema é otimizado para a administração de medições de séries temporais e suas “Tags” (etiquetas) associadas. Por Tags queremos dizer qualquer coisa usada para identificar mais informação gravada no sistema. Neste esquema isto inclui o nome de metadados, e o valor de metadados, que no esquema relacional seriam os atributos de qualquer Tabela. O UID é usado para gerenciar todas essas Tags diferentes, por isso temos o UID no nome da Tabela (tagsdb-uid), cada atributo (relacional) recebe o seu próprio ID único (UID) nesta Tabela.

A Tabela TAGSDB-UID é para gerenciamento de UIDs e utilizados em uma relação

de chave estrangeira da tabela TAGSDB, como já foi citado. Gravando um novo UID em duas linhas nesta tabela, uma Tag de mapeamento da NOME PARA UID, o outra é UID PARA A NOME, o nome da origem no nosso modelo. Por exemplo, gravar o pod-id “nimbus.LRG”, gera uma nova UID usada como *chave de linha* na *linha* UID PARA A NOME. A *coluna família* Nome para essa linha armazena a etiqueta Nome. O *qualificador da coluna* é usado como uma espécie de NAMESPACE - um delimitador abstrato (container) que fornece um contexto para os itens que ele armazena (nomes, termos técnicos, conceitos, etc), o que permite uma desambiguação para itens que possuem o mesmo nome mas que residem em espaços de nomes diferentes. Como um contexto distinto é fornecido para cada container, o significado de um nome pode variar de acordo com o espaço de nomes o qual ele pertence - para a UID, distinguindo este UID como um atributo (relacional) (por oposição à uma etiqueta de metadados do nome ou o valor). A *linha* da NOME PARA UID usa a Nome como a *chave de linha* e armazena o UID na *coluna família* ID, mais uma vez qualificada pelo tipo de Tag. O Código 5.2 a seguir mostra como usar o “script” TAGSDB para registrar dois novos atributos (etiquetas). Estas linhas também são usadas para mapear nomes de atributos (etiquetas) para seus UIDs associados ao gravar novos valores.

Código 5.2 – Registrando as métricas na tabela tagsdb-uid - autoria nossa

```
kimana@mbaka:~$ hbase shell
hbase(main):001:0> scan 'tagsdb-uid', {STARTROW => \0\0\1}
ROW                          COLUMN+CELL
 \x00\x00\x01                column=name:tags,     value=nimbus.LRG
 \x00\x00\x02                column=name:tags,     value=stratus.LRG
 nimbus.LRG                  column=id:tags,       value=\x00\x00\x01
 stratus.LRG                 column=id:tags,       value=\x00\x00\x02
4 row(s) in 0.0760 seconds
hbase(main):002:0>
```

5.2.1.2 Tabela TAGSDB

A Tabela TAGSDB é o coração da base de dados de séries temporais temporal intervalar: a Tabela que armazena séries temporais de *medições* e *metadados*. Esta tabela é projetada para suportar consultas destes dados filtrados por intervalo de timestamp

(timestamp final e inicial) e Tag (etiqueta), o que lhe dá a característica de uma BDT intervalar. Isto é realizado através de um projeto cuidadoso da *chave de linha*. A Tabela 4 ilustra a *chave de linha* para a tabela TAGSDB. Os UIDs gerados pelo registro Tag na Tabela TAGSDB-UID são usados aqui na *chave de linha* desta tabela. Esse esquema é otimizado para consultas centradas em atributos (Tags), de modo que o UID do atributo vem em primeiro lugar.

Tabela 4 – BNF da *chave de linha* da Tabela TAGSDB, autoria nossa.

UID Tag,Timestamp, Nome (tag),Valor (tag)

A Nome da Tag e valor UIDs estão por último na *chave de linha*. Armazenar todos esses atributos na *chave de linha* possibilita a filtragem dos resultados de pesquisa. O esquema contém apenas uma única *coluna família*, M. Isto é porque o HBase demanda pelo menos uma tabela conteúdo, ou seja, pelo menos uma *coluna família*. A tabela TAGSDB não usa a *coluna família* para organizar os dados, mas o Hbase necessita dela declarada.

5.3 EXPERIMENTO

O objetivo deste experimento é determinar o que tem melhor desempenho entre os bancos de dados MySQL® e HBase™, como já foi visto, mas é bom realçar neste experimento, as duas ferramentas Zenoss® e CloudStack™ utilizam seu armazenamento de dados em “single-node”, ou seja, uma arquitetura “standalone”, portanto a configuração aqui utilizada foi a mesma da implementação atual na Nuvem do LRG.

Apesar do teste ter sido feito em um “laptop” com boas configurações e não um server, este computador conta com um processador Intel® Core™ i7-2630QM, com 16GB de RAM DDR3 1333MHZ, Disco Rígido de 640GB com 7200rpm e suporte a virtualização assistida por hardware (HW).

5.3.1 Teste de Unidade

O HBase™ roda em cima do *Hadoop Distributed Filesystem (HDFS)*, fornecendo capacidades BigTable™ semelhantes para Hadoop™. Ambos têm suas próprias APIs, de modo que os usuários podem acessar seus bancos de dados com essas APIs em Java™ . A utilização do HBase™ em um nó só faz sentido em um ambiente de teste de unidade (que é este o caso). O HBase é realmente construído para grande volume de dados e

alta disponibilidade com uma configuração como esta a recomendação é utilizar um SGBD relacional como utilizado atualmente. Mas, como o propósito aqui é apenas verificar se a utilização do HBase™ faz sentido na configuração e utilização atual das ferramentas de Redes no ambito do LRG, foi feito o teste de unidade. Os benefícios do HBase™ passam sharding automático (autosharding ou particionamento horizontal), distribuição de carga, recuperação de nós falhos (failover), etc. Em termos práticos utilizar o HBase™ com menos de 10 nós geralmente é mesmo uma má idéia porque o *HDFS* não é otimizado de todo para esse tipo de configuração. Como foi possível notar do teste realizado, não se faz uso dos benefícios deste banco de dados para *Big Data* sem poder apreciar a sua velocidade, escalabilidade e confiabilidade.

5.3.2 Mapeamento de Tabelas MySQL® innoDB para HBase™

A principal limitação inicial do experimento foi pensar no mapeamento dos dados dos esquemas MySQL™ innoDB das ferramentas utilizadas no LRG para a forma como são armazenados no HBase™. A maneira mais limitada foi utilizar o Sqoop™, portanto os dados de uma tabela *SQL* indicando no seu comando a *Tabela*, *Coluna Família* e *Chave de Linha* como se verifica pelo Código 5.3.

Código 5.3 – Import com Sqoop™ para os dados de uma Tabela MySQL® para o HBase™

```
sqoop import -hbase-create-table -hbase-table alert --column-family info
-hbase-row-key id --connect jdbc:mysql://127.0.0.1/cloud --table alert -m 1
    --username root
```

Há também outras formas indiretas utilizando ferramentas como Hive™ e Pig™, mas será de forma muito artesanal e não fará com certeza utilização das maiores vantagens do HBase™, que é a utilização do modelo de dados com esquema proposto e em um contexto big data, pois infelizmente a quantidade de dados dos “dumps” dos esquemas das ferramentas no LRG ainda não acumularam dados o suficiente.

5.3.2.1 Tabelas Eventos e Alertas

A título de exemplificação das informações passiveis de se enquadrar no *esquema* mostrado anteriormente, foram utilizadas no experimento apenas duas (2) tabelas onde são armazenadas métricas Elas são salvas em *log* ou em banco de dados e, visualizáveis na *ferramenta de monitoramento* Zenoss ™.

Field	Type
uuid	binary(16)
fingerprint_hash	binary(20)
fingerprint	varchar(255)
status_id	tinyint(4)
event_group_id	int(11)
event_class_id	int(11)
event_class_key_id	int(11)
event_class_mapping_uuid	binary(16)
event_key_id	int(11)
severity_id	tinyint(4)
element_uuid	binary(16)
element_type_id	tinyint(4)
element_identifier	varchar(255)
element_title	varchar(255)
element_sub_uuid	binary(16)
element_sub_type_id	tinyint(4)
element_sub_identifier	varchar(255)
element_sub_title	varchar(255)
update_time	bigint(20)
first_seen	bigint(20)
status_change	bigint(20)
last_seen	bigint(20)
event_count	int(11)
monitor_id	int(11)
agent_id	int(11)
syslog_facility	int(11)
syslog_priority	tinyint(4)
nt_event_code	int(11)
current_user_uuid	binary(16)
current_user_name	varchar(32)
clear_fingerprint_hash	binary(20)
cleared_by_event_uuid	binary(16)
summary	varchar(255)
message	varchar(4096)
details_json	mediumtext
tags_json	mediumtext
notes_json	mediumtext
audit_json	mediumtext

Figura 27 – Tabela event summary ou event archive do Zenoss®, (CURRY, 2013).

No CloudStack™ existem duas tabelas: *event* e *alert* mostradas na Figura 28, no entanto todos os *alertas* do CloudStack™ são tratados como *eventos* pelo Zenoss®. Portanto, no fim são armazenados em duas tabelas, mas uma destinada ao arquivamento de eventos *event-archive* e outra àqueles que são sumarizados e mostrados pela ferramenta *event-summary*. A tabela pode ser vista na Figura 27. Para evitar a sobrecarga no CloudStack™ e Zenoss™, apenas os últimos dois (2) dias de eventos são verificados no Zenoss™. Isto permite a discrepância de fuso horário entre os servidores de ambos, bem como algum tempo de inatividade, ou sem eventos, (CURRY, 2013). Não há um mecanismo de tempo-real de série temporal na coleta de eventos com a API CloudStack, assim *alertas* e *eventos* só serão *monitorados uma vez por minuto*, entretanto, tal ocorrência não invalida

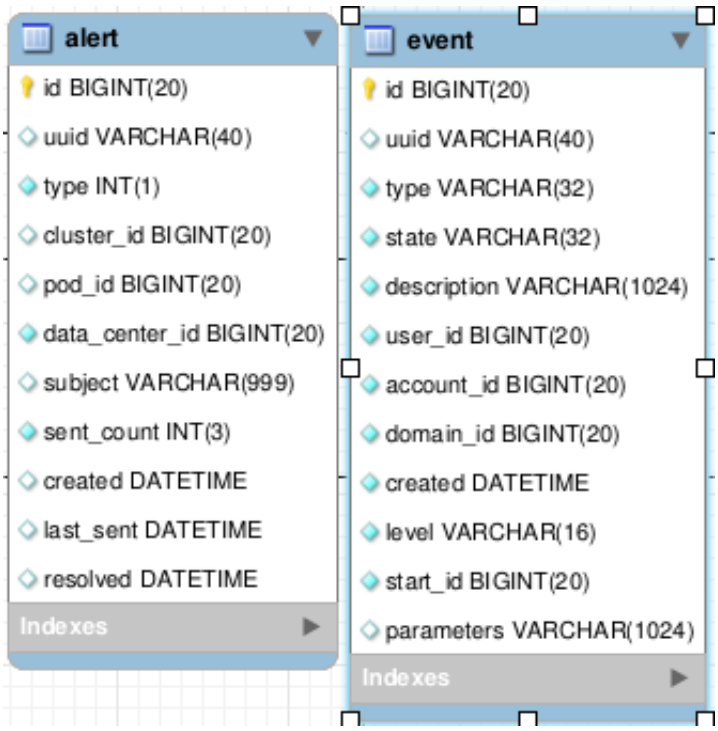

Figura 28 – Tabela event e alert do CloudStack®, elaborada pelo autor.

o experimento pois não está sendo testado o modelo proposto e sim as características de ambos SGBDs no estado atual de implementação na Nuvem.

5.3.3 Módulos

O experimento realizado é feito utilizando dois clientes implementados em linguagem Java™ que utilizam as APIs específicas e drivers para comunicação com o MySQL® e o Apache HBase™, é utilizado o gerenciador de projetos Apache Maven™ na *ambitente de desenvolvimento integrado (IDE)* Eclipse® para fazer a construção (build) de todas as dependências de bibliotecas (API). O que basicamente cada cliente faz é se ligar aos banco de dados e realizar transações elementares de *leitura (READ)*, *gravação (WRITE)* e *exclusão (DELETE)* utilizando os dados do "dump" para uma Tabela das ferramentas de Redes do LRG no caso do MySQL® e as Tabelas migradas para o Apache HBase™, estas foram "alert" e "event" como já mencionado.

Durante cada transação é calculado o tempo para a mesma em milisegundos (MS), pegando o tempo inicial START = SYSTEM.CURRENTTIMEMILLIS();, realizando a operação logo em seguida, e finalmente calculando este tempo de transação pela simples diferença entre o tempo do início e do fim da transação ELAPSED = SYSTEM.CURRENTTIMEMILLIS() - START;. Isto é feito percorrendo todos os registros das tabelas, que são em torno de

1000 dependendo da tabela utilizada e, depois, calculado o tempo da transação sobre eles em cada situação. No fim se obtêm a saída no console de todos tempos para cada operação nos respectivos registros. Estes de fato um grande volume de dados, mas para os propósitos deste teste, é o suficiente para verificar as vantagens do Apache HBase™ até neste ambiente muito simples.

5.3.3.1 MySQLTester

Conecta-se à tabela selecionada de um banco de dados MySQL® e testa-se o desempenho de *leitura (SELECT)*, *gravação (INSERT)* e *exclusão (DELETE)*. O cliente em Java™ não depende o *esquema da tabela*, e pode ser capaz de processar todo tipo de tabelas. Detecta automaticamente a chave primária e a usa para apagar e inserir um registro de/para a Tabela. O calculo do tempo é feito obtendo tempo inicial antes da transação e depois finalizado e calculada a diferença, portanto uma conta bem simples para o resultado convertido de tempo em UNIX para milesegundos em tipo longo. Um trecho do Código A.1 cliente é visto a seguir:

Código 5.4 – Testes de DELETE INSERT e READ no MySQLTester

```
// DELETE Test
sql = DELETE FROM + strTable + WHERE + keyName + =' + rsGuideline.getObject(keyName)
    + ';
start = System.currentTimeMillis();
stmt.execute(sql);
elapsed = System.currentTimeMillis() - start;
System.out.println(DELETE : + Long.toString(elapsed) + ms);

// INSERT Test
sql = INSERT INTO + strTable + VALUES(;
for (int i = 0; i < numberOfColumns; i++) {
        if (i > 0)
                sql += ,;
        sql += ?;
}
```

```
sql += );
pstmt = conn.prepareStatement(sql);
for (int i = 0; i < numberOfColumns; i++) {
        pstmt.setObject(i + 1, rsGuideline.getObject(i + 1));
}
start = System.currentTimeMillis();
pstmt.execute();
elapsed = System.currentTimeMillis() - start;
System.out.println(INSERT : + Long.toString(elapsed) + ms);

// SELECT Test
sql = SELECT * FROM + strTable + WHERE + keyName + =' +
    rsGuideline.getObject(keyName) + ';
start = System.currentTimeMillis();
stmt.execute(sql);
elapsed = System.currentTimeMillis() - start;
System.out.println(SELECT : + Long.toString(elapsed) + ms);
```

5.3.3.2 HBaseTester

Conecta-se à tabela selecionada de um banco de dados HBase™ e testa-se o desempenho de *leitura (GET)*, *gravação (PUT)* e *exclusão (DELETE)*. Este módulo também não depende do *esquema da tabela*, e pode ser capaz de processar todo tipo de tabelas. Detecta a *chave de linha* automáticamente e usa para apagar e inserir um registro a partir de/para a Tabela. O calculo do tempo é feito da mesma maneira que o cliente anterior. Um trecho do Código 5.5 cliente é visto a seguir:

Código 5.5 – Testes de DELETE INSERT e READ no HBaseTester

```
...
//DELETE Test
Delete d = new Delete(result.getRow());
start = System.currentTimeMillis();
table.delete(d);
```

```
elapsed = System.currentTimeMillis() - start;
System.out.println(DELETE : + Long.toString(elapsed) + ms);

//INSERT Test
HTableInterface addTable = connection.getTable(strTable);
Put put = new Put(result.getRow());
for (KeyValue kv : result.raw()) {
        put.add(kv.getFamily(), kv.getQualifier(), kv.getValue());
}
start = System.currentTimeMillis();
addTable.put(put);
elapsed = System.currentTimeMillis() - start;
System.out.println(INSERT : + Long.toString(elapsed) + ms);

//READ Test
Get get = new Get(result.getRow());
start = System.currentTimeMillis();
Result rsGet = addTable.get(get);
elapsed = System.currentTimeMillis() - start;
System.out.println(GET : + Long.toString(elapsed) + ms);
...
```

5.3.3.3 Análise dos Resultados

A partir dos Gráficos nas Figuras 29, 30, 31 criados com os dados obtidos na saída da execução dos clientes, é possivel concluir que desempenho de HBase™ é muito melhor que o do MySQL® em ESCRITA e EXCLUSÃO. Na operação de LEITURA, o Apache HBase™ é semelhante ao MySQL®, mas HBase tem ainda melhor curva de desempenho enquanto no MySQL o desempenho varia de forma aleatória. Se for executado o mesmo teste várias vezes sobre os mesmos dados e mesma tabela (alert ou event) a saída permanece semelhante. De qualquer forma, o desempenho médio dos HBase® é melhor do que do MySQL®, mas nesta arquitetura utilizada não tira proveito da escalabilidade do Apache HBase® para LEITURAS e muito mesmo no contexto Big Data, pois os dados não são o

suficientes, entretanto a título de criar um primeiro protótipo da migração de SGDB para o outro, o SGBD NoSQL mostrou-se extremamente eficiente mesmo sem a utilização do modelo de dados proposto neste trabalho.

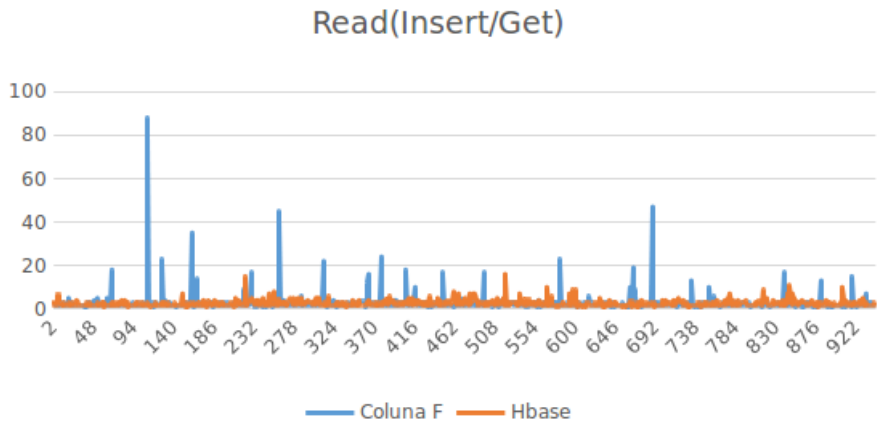

Figura 29 – Desempenho comparativo para LEITURA do HBase® e MySQL™.

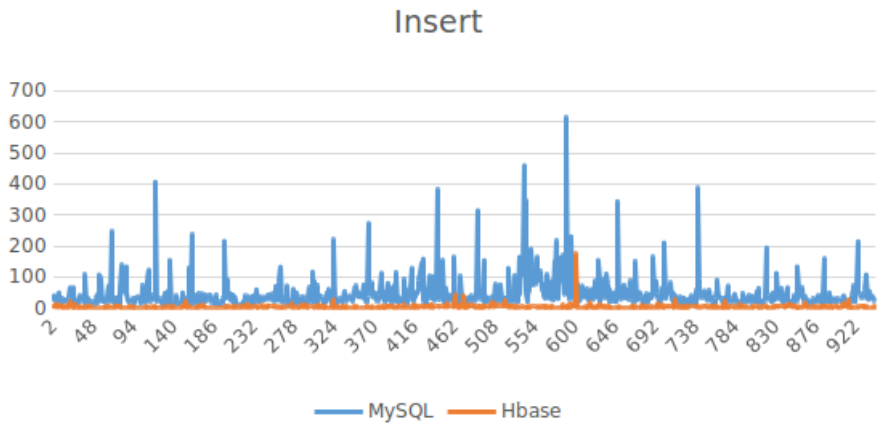

Figura 30 – Desempenho comparativo para ESCRITA do HBase® e MySQL™.

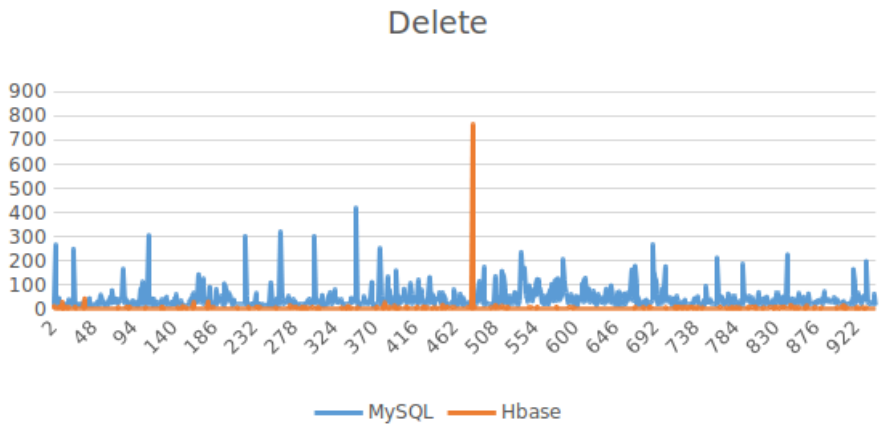

Figura 31 – Desempenho comparativo para EXCLUSÃO do HBase® e MySQL™.

6 CONCLUSÃO E TRABALHOS FUTUROS

6.1 CONCLUSÃO

Para finalizar esta obra, podemos concluir que foi apresentado os pontos pertinentes de todos conceitos teóricos envolvidos na Arquitetura Cognitivo-Autônoma de Monitoramento em Nuvem, destacando os conceitos mais aderentes, principalmente: Computação em Nuvem e sua correlação com a Computação Autônoma e Cognitiva.

Foi analisado dentro do contexto da mesma Arquitetura o ferramental de Gerência e Monitoramente de Nuvens e sua real aderência às propriedades indicadas pelo autor de referência, no estado da arte da avaliação deste tema, uma vez que o mesmo não têm sido alvo de uma análise mais adequada e minuciosa, com isso e associado à correlação com o projeto da Arquitetura de Monitoramento, este autor conclui que as ferramentas escolhidas no âmbito de relatório de pesquisa LRG o qual colabora estão alinhadas à área de pesquisa que trata o tema deste trabalho que é armazenamento de informações de monitoramento em Nuvem, pois tanto o CloudStack™ e o Zenoss ZenPack™ segundo Aceto et al. (2013) possuem a propriedade de TIMELINESS (PONTUALIDADE), a qual indica que os eventos dectáveis da Nuvem estão disponíveis em tempo de utilização, o que para a tomada de decisões com base na mineração de dados e, utilização de algoritmos de descoberta de conhecimento utilizando relações temporais para descobrir novos padrões de comportamento do sistema, bem como relações de causalidade que não poderiam ser estabelecidas apenas por um registro atômico da ocorrência de eventos, é primordial.

Quanto a comparação de banco de dados NoSQL e SQL: Primeiramente, foi explorada as vantagens de um tipo de SGBD sobre o outro no contexto de grande volume de dados enumeradando as qualidades do NoSQL como a adjacência livre de índice, o esquema dinâmico acessado via API e a escalabilidade horizontal por replicação que contorna problemas de “failover”, como também a não aderência as propriedade ACID em decorrência da prioridade por escalabilidade horizontal. Depois foi contextualizado o porquê da utilização do Apache HBase™ na solução do modelo de dados intervalar-temporal em séries temporais. Por fim, foi feito um experimento com clientes em Java para Oracle MySQL® e Apache HBase™ que realizaram operações de EXCLUSÃO, ESCRITA e

LEITURA (esta última muito importante para a disponibilização de dados em real-time das informações da Nuvem) e, foi comprovado que o HBase™ é melhor em todas operações em uma instalação “standalone” exceto em relação ao READ que ambas tiveram quase o mesmo empenho, exceto pela inconstância do BD relacional. Portanto, a observação principal da experiência fica para o fato de a curva do MySQL® ser muito inconstante, em que em algumas leituras houve uma grande latência mesmo executanto o teste de unidade com os clientes mais de uma vez. Foi tratada também a possibilidade de utilização de séries-temporais pelo HBase™ ser mais eficiente do que qualquer outro banco de dados relacional, uma vez que estes para armazenar grandes quantidade de dados precisam de fazer um particionamento vertical espalhando os dados, o que numa operação de READ seria problemático e com grande latência. Os tipos de dados complexos em uma série temporal não seriam adequados para o armazenamento em MySQL pois utilizariam tipos complexos como “blobs". A aderência do HBase™ à propriedade CP do teorema CAP, o torna consistênte, o que facilita a visualização de “dashboards” no caso de utilização de ferramentas de monitoramento e gerência e possibilidade de geração de relatórios em tempo real. A questão mais crítica para a utilização de MySQL® innoDB é o fato que depois do acumulo de dados em meses em pontos de dados de séries temporais sua aplicação se torna inviável, portanto em termos de custo e complexidade da utilização do MySQL® para os mesmos fins é preferivel utilizar o HBase™.

A escolha das informações para análise do modelo não foi ao acaso, trata-se de métricas (alertas e eventos) que não devolvem informações em tempo real ao dashboard das duas ferramentas implementadas no âmbito do LRG. Foi escolhido assim pois não existia a certeza de que o modelo de dados inicialmente proposto iria partir para uma modelagem real-time com inclusão dos benefícios de time-series (grande volume de dados em sequência e estrutura mais complexa para armazená-los (tempo,valor)). Portanto, todo estudo feito em termos de levantamento teórico neste trabalho e solução encontrada, foi feita de forma sistemática e na tentativa de pormenorizar o máximo póssivel todos pontos relevantes:

O primeiro passo, foi realizar o experimento apresentado neste trabalho onde verificou-se a eficiência do HBase™ e em termos de esquema a ferramenta Sqoop™ facilita o trabalho não sendo necessário um mapeamento minucioso das tabelas e atributos das mesmas para o modelo lógico do Apache HBase™. O segundo passo foi realizar de fato

um mapeamento de uma tabela relacional para o modelo NoSQL sem recorrer a nenhuma ferramenta, o que se pode comprovar é que o modelo de dados proposto com o esquema utilizado neste trabalho acomoda adequadamente todos os tipos de dados pois é flexível o suficiente para tal, sem deixar de lado qualquer tipo de informação existente nas tabelas vistas e, portanto, o mesmo é valido para qualquer Tabela pois o modelo lógico do HBase™ beneficia a inclusão de informação nas linhas e o tratamento de qualquer outro tipo de atributo como “tag” (etiqueta) minimiza problemas de acomodação dos dados mapeados de um paradigma para o outro. O terceiro e último passo foi finalmente propor o Modelo de Dados temporal intervalar após a consulta de vasta bibliografia e, possível de ser aplicado uma vez que este ambiente é pensado para um grande volume de dados advindos das informações de monitoramento de uma Nuvem.

Com isso tudo foi possível definir uma chave de linha considerando os gargalos encontrados por desenvolvedores e na literatura, principalmente o problema do lock-step onde dois clientes concorrem pela mesma região no HFile HBase™ e, enfrenta um crescimento monotônico no caso de utilizar a timestamp na primeira posição de uma “row-key”. A solução encontrada na literatura foi bastante simples e, pareceu um passe de mágica, mas é aplicada em algumas soluções sólidas que realizam o monitoramento de redes com implementação em séries temporais que é o caso do OpenTSDB™.

Considera-se que o presente trabalho contribuiu satisfatoriamente para o que foi definido como objetivo principal, objetivos específicos, e os estudos preliminares alcançados. Portanto, existe um amplo leque trabalhos futuros a serem desenvolvidos na mineração de dados, algoritmos de conhecimento para tratar da arquitetura de monitoramento, computação autonômica e desenvolvimento da mesma arquitetura, integrando o armazenamento de dados *NoSQL* em Apache HBase™ e realização de consultas temporais intervalares com vista a tomar decisões importantes na gerência de uma Nuvem.

6.2 TRABALHOS FUTUROS

Em relação aos trabalhos futuros, o primeiro passo seria a implementação de fato do modelo de dados proposto, partindo disso seria possível pensar em várias formas de se implementar o coletor de dados na ACAMN. Toda a solução considera que estamos tratando de um grande volume de dados e por isso a opção por séries temporais no monitoramento,

mas em termos de protótipo poderia ser feita uma grande simplificação apenas com a coleta, seleção e etiquetagem dos dados em um modo de implementação mais simples (single-node, master-slave) do Apache HBase™ se beneficiando de apenas algumas vantagens como o esquema dinâmico e as escritas atômicas, para depois estender de forma mais sistemática o protótipo iniciado.

Com o coletor de dados definido e o empacotamento de dados, era possivel propor uma otimização do esquema de dados que consumisse menos “bytes” no armazenamento do HBase™, visto que quanto menor em bytes forem os campos, torna-se viável uma maior inserção de dados, maior compactação dos mesmos. Entretanto, ainda eria necessário pensar em denormalização de tabelas e consequente repetição de índices artificiais (chaves estrangeiras), porque nada supera esta generalização para acomodar toda quantidade possível de um esquema realmente complexo de dados em SQL e as leituras privilegiadas com as chaves o mais complexas possíveis, leituras estas utilizadas para fazer inferências nas decisões do gerênciamento da Nuvem. Finalmente, ainda há uma gama enorme de aplicações a serem pensadas, o tema não é inesgotável mas o leque de possibilidades e adaptações é muito grande.

REFERÊNCIAS

ACETO, G. et al. Survey cloud monitoring: A survey. **Computer Networks.**, Elsevier North-Holland, Inc., New York, NY, USA, v. 57, n. 9, p. 2093–2115, 2013. ISSN 1389-1286.

BADGER, M. **Zenoss Core Network and System Monitoring**. Packt Publishing, Limited, 2008. ISBN 9781847194299. Disponível em: <http://books.google.com.br/books?id=B3YBMfU_u8sC>.

BERG, K.; SEYMOUR, T.; GOEL, R. History of databases. **International Journal of Management & Information Systems**, v. 17, n. 1, 2013.

BESS, C. **Is Big Data Ousting Cloud Computing as this Year's Buzzword?** 2011. Acessado em Julho de 2014. Disponível em: <http://www.enterprisecioforum.com/en/blogs/cebess/big-data-ousting-cloud-computing-year%E2%80%99s>.

BEYNON-DAVIES, P. **Database Systems**. [S.l.]: Palgrave Macmillan Limited, 2004. (Macmillan computer science series). ISBN 9781403916013.

BRISCOE, G.; MARINOS, A. Digital ecosystems in the clouds: Towards community cloud computing. **CoRR**, abs/0903.0694, 2009.

BUYYA, R.; BROBERG, J.; GOSCINSKI, A. M. **Cloud Computing Principles and Paradigms**. [S.l.]: Wiley Publishing, 2011. ISBN 9780470887998.

BUYYA, R.; CALHEIROS, R. N.; LI, X. Autonomic cloud computing: Open challenges and architectural elements. **CoRR**, abs/1209.3356, 2012.

BUYYA, R. et al. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. **Future Generation Computer Systems**, v. 25, n. 6, p. 599 – 616, 2009. ISSN 0167-739X.

CANTU, A. **The History and Future of Cloud Computing**. 2011. Acesso em Maio de 2014. Disponível em: <http://www.forbes.com/sites/dell/2011/12/20/the-history-and-future-of-cloud-computing/>.

CHANTRY, D. **Mapping Applications to the Cloud**. 2009. Acesso em Maio de 2014. Disponível em: <http://msdn.microsoft.com/en-us/library/dd430340.aspx>.

CHAVES, S. D.; URIARTE, R.; WESTPHALL, C. Toward an architecture for monitoring private clouds. **Communications Magazine, IEEE**, v. 49, n. 12, p. 130–137, Dezembro 2011. ISSN 0163-6804.

CHIFU, V. et al. Semantic web service composition method based on fluent calculus. In: **Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009 11th International Symposium on**. [S.l.: s.n.], 2009. p. 325–332.

CHISNALL, D. **The Definitive Guide to the Xen Hypervisor**. [S.l.]: Prentice Hall, 2008. (Prentice Hall open source software development series). ISBN 9780132349710.

CURRY, J. **Event Management for Zenoss Core 4**. 2013. 152 p.

DIMIDUK, N. et al. **HBase in Action**. [S.l.]: Manning Publications Company, 2012. (Running Series). ISBN 9781617290527.

DOBSON, S. et al. Fulfilling the vision of autonomic computing. **Computer (New York)**, v. 43, n. 1, p. 35–41, 2010.

DYER, R. **Mysql in a Nutshell**. Second. [S.l.]: O'Reilly, 2008. ISBN 9780596514334.

ELMASRI, R. A.; NAVATHE, S. B. **Fundamentals of Database Systems**. 3rd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0805317554.

EMERSON, S. L.; DARNOVSKY, M.; BOWMAN, J. **The Practical SQL Handbook: Using Structured Query Language**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0-201-51738-8.

ENDO, P. T.; SADOK, D.; KELNER, J. Autonomic cloud computing: Giving intelligence to simpleton nodes. **2013 IEEE 5th International Conference on Cloud Computing Technology and Science**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 502–505, 2011.

FOSTER, I.; KESSELMAN, C. **The Grid 2: Blueprint for a New Computing Infrastructure**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558609334.

FOSTER, I. T. et al. Cloud computing and grid computing 360-degree compared. **CoRR**, abs/0901.0131, 2009.

GEORGE, L. **HBase - The Definitive Guide: Random Access to Your Planet-Size Data**. [S.l.]: O'Reilly, 2011. I-XXVII, 1-522 p. ISBN 978-1-449-39610-7.

HADJIGEORGIOU, C. **RDBMS vs NoSQL - Performance and Scaling Comparison**. Dissertação (Mestrado) — University of Edinburgh, 2013. Disponível em: <https://www.epcc.ed.ac.uk/sites/default/files/Dissertations/2012-2013/RDBMS%20vs%20NoSQL%20-%20Performance%20and%20Scaling%20Comparison.pdf>.

HBASECOMPRESSION. **The Apache HBase™ Reference Guide - Appendix C- Compression in HBase**. 2014. Acessado em Junho de 2014. Disponível em: <http://hbase.apache.org/book/lzo.compression.html>.

HBASEGUIDE. **The Apache HBase™ Reference Guide - Schema Design**. 2014. Acessado em Abril de 2014. Disponível em: <http://hbase.apache.org/book/schema.casestudies.html>.

HBASEUSERLIST. **Lista de discussões de usuários do HBase**. 2010. Acessado em Abril de 2014. Disponível em: <http://grokbase.com/t/hbase/user/10ax44ffw7/time-series-schema>.

IYER, R. et al. Virtual platform architectures for resource metering in datacenters. **SIGMETRICS Perform. Eval. Rev.**, ACM, New York, NY, USA, v. 37, n. 2, p. 89–90, 2009. ISSN 0163-5999. Disponível em: <http://doi.acm.org/10.1145/1639562.1639600>.

KAI, L. et al. Scm: A design and implementation of monitoring system for cloudstack. In: **Proceedings of the 2013 International Conference on Cloud and Service Computing**. Washington, DC, USA: IEEE Computer Society, 2013. (CSC '13), p. 146–151. ISBN 978-0-7695-5157-9. Disponível em: <http://dx.doi.org/10.1109/CSC.2013.30>.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 1, p. 41–50, 2003. ISSN 0018-9162.

KHURANA, A. **Building Applications with HBase**. 2013. Acesso em Março de 2014. Apresentação de Power Point. Disponível em: <http://pt.slideshare.net/amansk/building-apps-with-h-base-data-days-texas-march-2013>.

KING, D.; FORD, C. A critical survey of network functions virtualization (nfv). In: ____. **iPOP**. [S.l.: s.n.], 2013.

KUMAR, P. **Cloud Database Schema – 2.2.14**. 2013. Acesso em Junho de 2014. Disponível em: <http://www.docstoc.com/docs/146869941/Database-SChema---CloudStack>.

KUTARE, M. et al. Monalytics: Online monitoring and analytics for managing large scale data centers. In: **Proceedings of the 7th International Conference on Autonomic Computing**. New York, NY, USA: ACM, 2010. (ICAC '10), p. 141–150. ISBN 978-1-4503-0074-2.

MAPPIC, S. **Big Data Monitoring**. 2014. Acessado em Janeiro de 2014. Disponível em: <http://sandimappic.ulitzer.com/node/3003723>.

MEI, L.; CHAN, W. K.; TSE, T. H. A tale of clouds: Paradigm comparisons and some thoughts on research issues. In: **Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference**. Washington, DC, USA: IEEE Computer Society, 2008. (APSCC '08), p. 464–469. ISBN 978-0-7695-3473-2.

MELL, P. M.; GRANCE, T. **SP 800-145. The NIST Definition of Cloud Computing**. Gaithersburg, MD, United States, 2011.

MESSINGER, L. Better explaining the cap theorem. **Dzone**, p. 2, Fevereiro 2013. Disponível em: <http://architects.dzone.com/articles/better-explaining-cap-theorem>.

MITSA, T. **Temporal Data Mining**. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2010. ISBN 1420089765, 9781420089769.

MYSQL. **Mysql website**. 2014. Acessado em Março de 2014. Disponível em: <http://www.mysql.com/>.

OLIVEIRA, C. **SQL - Curso Prático**. [S.l.]: Novatec, 2002. ISBN 9788575220245.

OPENTSDB. **Documentation for OpenTSDB 2.0**. 2014. Acessado em Maio de 2014. Disponível em: <http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html>.

PALHARES, N. **Uma abordagem estratificada à monitorização de serviços Cloud**. Dissertação (Mestrado) — Universidade do Minho - Departamento de Informática, Dezembro 2012. Disponível em: <http://hdl.handle.net/1822/27952>.

PITANGA, M. **Computação em cluster: o estado da arte da computação**. [S.l.]: Brasport, 2004. ISBN 9788574521565.

PRASAD, S.; AVINASH, S. Smart meter data analytics using opentsdb and hadoop. In: **Innovative Smart Grid Technologies - Asia (ISGT Asia), 2013 IEEE**. [S.l.: s.n.], 2013. p. 1–6.

RAFIEI, D.; MENDELZON, A. Similarity-based queries for time series data. In: ACM. **ACM SIGMOD Record**. [S.l.], 1997. v. 26, n. 2, p. 13–25.

SABHARWAL, N.; SHANKAR, R. **Apache CloudStack cloud computing: leverage the power of CloudStack and learn to extend the CloudStack environment**. Birmingham: Packt Publ., 2013. (Community experience distilled).

SADALAGE, P. J.; FOWLER, M. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 0321826620, 9780321826626.

SCHUBERT, F. **Uma Arquitetura de Computação Autônoma e Cognitiva para Monitoramento de Nuvens**. Dissertação (Dissertação de Mestrado) — INE - Departamento de Informática e Estatística - Universidade Federal de Santa Catarina (UFSC), Agosto 2014. Não publicado.

SCHWALB, E.; VILA, L. Temporal constraints: A survey. **Constraints**, Kluwer Academic Publishers, Hingham, MA, USA, v. 3, n. 2/3, p. 129–149, jun. 1998. ISSN 1383-7133. Disponível em: <http://dx.doi.org/10.1023/A:1009717525330>.

SCHWARTZ, B. et al. **High Performance Mysql, 2Nd Edition**. Second. [S.l.]: O'Reilly, 2008. ISBN 9780596101718.

SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. **Database Systems Concepts**. 5. ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN 0072958863, 9780072958867.

SOTOMAYOR, B. et al. Virtual infrastructure management in private and hybrid clouds. **IEEE Internet Computing**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 13, n. 5, p. 14–22, 2009. ISSN 1089-7801.

SPRING, J. Monitoring cloud computing by layer, part 1. **IEEE Security & Privacy**, v. 9, n. 2, p. 66–68, 2011.

STONEBRAKER, M. Sql databases v. nosql databases. **Commun. ACM**, ACM, New York, NY, USA, v. 53, n. 4, p. 10–11, 2010. ISSN 0001-0782.

WIKIZENOSS. **WIKIZENOSS**. 2014. Acessado em Abril de 2014. Disponível em: <http://wiki.zenoss.org/>.

XENSERVER. **Xen Server Documentation**. 2014. Acessado em Março de 2014. Tradução nossa. Disponível em: <http://support.citrix.com/article/CTX137836>.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of Internet Services and Applications**, Springer-Verlag, v. 1, n. 1, p. 7–18, 2010. ISSN 1867-4828.

APÊNDICE A – PRIMEIRO APÊNDICE

Código A.1 – Cliente de testes de DELETE INSERT e READ no MySQLTester

```
/**
 *
 */
package rwork.lao.hbase;

import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @author maxwell.cco@gmail.com
 *
 */
public class MysqlTester {

        static final String strWelcomeMsg = "Welcome to MySQL Tester projects.";
        static final String strCopyrightMsg = "Created by maxwell.cco@gmail.com";
        static final String strGoodByeMsg = "Good Bye!";

        static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
        static final String DB_URL = "jdbc:mysql://localhost:3306/cloud";

        private String strServer = "localhost";
        private String strDatabase = "cloud";
```

```
private String strTable = "alert";

private String strUserName = "root";
private String strPassword = "m0z1n9x2";

private Connection conn = null;

/**
 *
 */
public MysqlTester() {
}

/**
 * @param args
 */
public static void main(String[] args) {
        System.out.println(strWelcomeMsg);
        System.out.println(strCopyrightMsg);

        if (args.length < 3) {
                usage();
        }

        MysqlTester instance = new MysqlTester();
        instance.exec(args);

        System.out.println(strGoodByeMsg);
}

public static void usage() {
        System.out.println("MysqlTester server_addr database_name table_name
            [UserName [Password]]");
```

```
}

private void parseArgs(String[] args) {
        if (args.length > 0)
                strServer = args[0];
        if (args.length > 1)
                strDatabase = args[1];
        if (args.length > 2)
                strTable = args[2];
        if (args.length > 3)
                strUserName = args[3];
        if (args.length > 4)
                strPassword = args[4];
}

private void exec(String[] args) {
        parseArgs(args);
        doTest();
}

private void doTest() {
        System.out.printf("Connecting to database [%s/%s]...\n", strServer,
            strDatabase);
        getConnection();
        if (conn == null) {
                System.out.println("Database connection failed.");
                return;
        }
        doTestLogic();
        closeConnection();
}

private void getConnection() {
```

```
        if (conn != null)
                return;
        conn = null;
        try {
                Class.forName(JDBC_DRIVER);
                conn = DriverManager.getConnection(String.format(DB_URL,
                    strServer, strDatabase), strUserName, strPassword);
        } catch (ClassNotFoundException e) {
                System.err.println("Database Driver is not installed. Check
                    class path, please.");
        } catch (SQLException e) {
                System.err.println(e.getMessage());
        } finally {
        }
}

private void closeConnection() {
        try {
                if (conn != null)
                        conn.close();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}

private void doTestLogic() {
        Statement stmt = null;
        ResultSet rs = null;
        try {
                System.out.println("Collecting record key values...");
                stmt = conn.createStatement();

                String sql;
```

```
                sql = "SELECT * FROM " + strTable;

                rs = stmt.executeQuery(sql);

                System.out.printf("Dump table [%s] in database [%s] ...\n",
                    strTable, strDatabase);
                ResultSetMetaData rsMetaData = rs.getMetaData();
                int numberOfColumns = rsMetaData.getColumnCount();
                for (int i = 0; i < numberOfColumns; i++) {
                        String fieldName = rsMetaData.getColumnName(i + 1);
                        if (i > 0)
                                System.out.print(", ");
                        System.out.print(fieldName);
                }
                System.out.println();
                while (rs.next()) {
                        doTestItem(rs);
                }
        } catch (SQLException e) {
                System.err.println(e.getMessage());
        } finally {
                try {
                        if (stmt != null)
                                stmt.close();
                } catch (SQLException e) {}
        }
}

private void doTestItem(ResultSet rsGuideline) {
        Statement stmt = null;
        ResultSet rs = null;
        PreparedStatement pstmt = null;
        try {
```

```
stmt = conn.createStatement();

// Prepare key column
DatabaseMetaData dm = conn.getMetaData();
rs = dm.getPrimaryKeys(null, null, strTable);
String keyName = "id";
if (rs.next()) {
        keyName = rs.getString("COLUMN_NAME");
}

// Dump record data
ResultSetMetaData rsMetaData = rsGuideline.getMetaData();
int numberOfColumns = rsMetaData.getColumnCount();
for (int i = 0; i < numberOfColumns; i++) {
        if (i > 0)
                System.out.print(", ");
        System.out.print(rsGuideline.getObject(i + 1));
}
System.out.println();

String sql;
long start, elapsed;

// DELETE Test
sql = "DELETE FROM " + strTable + " WHERE " + keyName + "='" +
    rsGuideline.getObject(keyName) + "'";
start = System.currentTimeMillis();
stmt.execute(sql);
elapsed = System.currentTimeMillis() - start;
System.out.println("DELETE : " + Long.toString(elapsed) + "
    ms");

// INSERT Test
```

```
                sql = "INSERT INTO " + strTable + " VALUES(";
                for (int i = 0; i < numberOfColumns; i++) {
                        if (i > 0)
                                sql += ",";
                        sql += "?";
                }
                sql += ")";
                pstmt = conn.prepareStatement(sql);
                for (int i = 0; i < numberOfColumns; i++) {
                        pstmt.setObject(i + 1, rsGuideline.getObject(i + 1));
                }
                start = System.currentTimeMillis();
                pstmt.execute();
                elapsed = System.currentTimeMillis() - start;
                System.out.println("INSERT : " + Long.toString(elapsed) + "
                    ms");

                // SELECT Test
                sql = "SELECT * FROM " + strTable + " WHERE " + keyName + "='"
                    + rsGuideline.getObject(keyName) + "'";
                start = System.currentTimeMillis();
                stmt.execute(sql);
                elapsed = System.currentTimeMillis() - start;
                System.out.println("SELECT : " + Long.toString(elapsed) + "
                    ms");

        } catch (SQLException e) {
                System.err.println(e.getMessage());
        } finally {
                try {
                        if (stmt != null)
                                stmt.close();
                } catch (SQLException e) {}
```

```
                    try {
                        if (pstmt != null)
                            pstmt.close();
                    } catch (SQLException e) {}
                }
        }

}
```

APÊNDICE B – SEGUNDO APÊNDICE

Código B.1 – Cliente de testes de DELETE INSERT e READ no HBaseTester

```
/**
 *
 */
package mysql;

import java.io.IOException;
import java.util.ArrayList;
import java.util.NavigableMap;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HConnection;
import org.apache.hadoop.hbase.client.HConnectionManager;
import org.apache.hadoop.hbase.client.HTableInterface;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;



/**
 * @author maxwell.cco@gmail.com
```

```
 *
 */
public class HBaseTester {

	public static final String HBASE_CONFIGURATION_ZOOKEEPER_QUORUM =
	"hbase.zookeeper.quorum";
	public static final String HBASE_CONFIGURATION_ZOOKEEPER_CLIENTPORT =
	"hbase.zookeeper.property.clientPort";

	static final String strWelcomeMsg = "Welcome to HBase Tester projects.";
	static final String strCopyrightMsg = "Created by maxwell.cco@gmail.com";
	static final String strGoodByeMsg = "Good Bye!";

	private String strTable = "event";
	private String strUserName = "root";
	private String strPassword = "m0z1n9x2";

	private Configuration config = null;
	private HConnection connection = null;


	public HBaseTester() { }

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		System.out.println(strWelcomeMsg);
		System.out.println(strCopyrightMsg);

		if (args.length < 1) {
			usage();
		}
```

```
        HBaseTester instance = new HBaseTester();
        instance.exec(args);

        System.out.println(strGoodByeMsg);
}

public static void usage() {
        System.out.println("HBaseTester table_name");
}

private void parseArgs(String[] args) {
        if (args.length > 0)
                strTable = args[0];
        if (args.length > 2)
                strUserName = args[2];
        if (args.length > 3)
                strPassword = args[3];
}

private void exec(String[] args) {
        parseArgs(args);
        doTest();
}

private void doTest() {
        config = HBaseConfiguration.create();
        HTableInterface table = null;

        try {
                connection = HConnectionManager.createConnection(config);
                table = connection.getTable(strTable);
```

```
                System.out.printf("Dump table [%s] ...\n", strTable);

                Scan scan = new Scan();
                ResultScanner scanner = table.getScanner(scan);
                boolean isFirst = true;
                ArrayList<String> columns = null;
                for (Result result : scanner) {
                        if (isFirst) {
                                isFirst = false;
                        }
                        doTestItem(table, result, columns);
                }
        } catch (IOException e) {
                e.printStackTrace();
                System.err.println(e.getMessage());
                return;
        } finally {
                try {
                        if (table != null)
                                table.close();
                } catch (IOException e) {}
        }
}

private void doTestItem(HTableInterface table, Result result,
    ArrayList<String> columns) {
        String key = Bytes.toString(result.getRow());
        System.out.print(key);

        for (KeyValue kv : result.raw()) {
                System.out.print(", " + Bytes.toString(kv.getQualifier()) +
                    "[" + Bytes.toString(kv.getValue()) + "]");
        }
```

```
System.out.println();

try {
        long start, elapsed;

        //DELETE Test
        Delete d = new Delete(result.getRow());
        start = System.currentTimeMillis();
        table.delete(d);
        elapsed = System.currentTimeMillis() - start;
        System.out.println("DELETE : " + Long.toString(elapsed) + "
            ms");

        //INSERT Test
        HTableInterface addTable = connection.getTable(strTable);
        Put put = new Put(result.getRow());
        for (KeyValue kv : result.raw()) {
                put.add(kv.getFamily(), kv.getQualifier(),
                    kv.getValue());
        }
        start = System.currentTimeMillis();
        addTable.put(put);
        elapsed = System.currentTimeMillis() - start;
        System.out.println("INSERT : " + Long.toString(elapsed) + "
            ms");

        //READ Test
        Get get = new Get(result.getRow());
        start = System.currentTimeMillis();
        Result rsGet = addTable.get(get);
        elapsed = System.currentTimeMillis() - start;
        System.out.println("GET : " + Long.toString(elapsed) + " ms");
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

}
```