Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

ANÁLISE COMPARATIVA SOBRE A DISPONIBILIDADE DE BANCO DE DADOS NAS NUVENS

Trabalho de Graduação

Edson Barboza de Lima

Recife, 24 de novembro de 2015

Universidade Federal de Pernambuco

Centro de Informática

Edson Barboza de Lima

ANÁLISE COMPARATIVA SOBRE A DISPONIBILIDADE DE BANCO DE DADOS NAS NUVENS

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Fernando da Fonseca de Souza

Recife, 12 de janeiro de 2016

Dedico este trabalho a Jozemil Barboza e Lenita Freitas, meus amados pais

RESUMO

A utilização de banco de dados em aplicações, particularmente em ambiente na nuvem é uma das formas para registrar e armazenar informações dos usuários de maneira confiável e segura. Cada vez mais empresas do ramo de informática investem fortemente nesse tipo de tecnologia a fim de fornecer ao cliente o melhor método de armazenar seus dados promovendo acesso e recuperação simples. Porém, a forma como esta informação é gerenciada segue uma grande variedade de restrições, sendo preciso tratar novos aspectos para conseguir ultrapassar essas barreiras e prover a informação de forma apropriada.

De acordo com o teorema CAP (Brewer 2012), um sistema distribuído em grande escala é capaz de fornecer simultaneamente apenas duas das três características:

1. Consistência (C de *Consistency*);
2. Disponibilidade (A de *Availability*); e
3. Tolerância à partição da rede (após ocorrer uma falha em uma das partes do sistema, este continuar funcionando) (P de *Partition tolerance*).

Com essa restrição, tecnologias de Banco de Dados (BD) em nuvem, em sua maioria, optam por fornecer maior Disponibilidade e Tolerância à partição, relaxando aspectos de consistência. Cada sistema de banco de dados em nuvem garante um nível de consistência diferente ou configurável, sendo preciso definir métodos e políticas específicos. Este trabalho tem por objetivo estabelecer índices que indiquem a motivação por parte de aplicações da escolha desses sistemas de banco de dados em nuvem. Será realizada uma análise das técnicas empregadas por tais sistemas que afetam diretamente as características CAP, em caso especial a disponibilidade. Por fim, o trabalho deve expor um experimento de consistência e disponibilidade do sistema Cassandra o qual caracteriza por sua flexibilidade e dinâmica no gerenciamento de grandes volumes de dados.

Palavras-chave: CAP, NoSQL, Sistema Distribuído, Escalabilidade, Balanceamento, Tolerância a Falha, Partição da Rede

ABSTRACT

Using databases in applications, particularly in cloud environment is one of the ways to record and store information from users reliably and securely. Increasingly information technology companies invest heavily in such paradigm in order to provide their customers the best way to store data and to allow easy access and retrieval. However, the way such information is managed faces a variety of restrictions; therefore it is needed to address new aspects to overcome these barriers in order to provide information adequately.

According to the CAP theorem (Brewer et al 2012), a distributed system on a large scale can simultaneously provide two of the three features, namely:

1. Availability;
2. Consistency; and
3. Partition Tolerance.

With this restriction, cloud database technologies, mostly choose to provide greater availability and partition tolerance, relaxing consistency aspects. Each cloud database system provides a different level of consistency or even a configurable one, being necessary to define specific methods and policies. This work aims at establishing rates to indicate the motivation of applications to choose a given cloud database system, by conducting an analysis of the techniques employed by such systems that directly may affect the CAP features.

Key words: CAP, Distributed System, Partition Tolerance, Scalability, NoSQL, Network Partition

LISTA DE SIGLAS E ABREVIATURAS

SGBD – Sistema de Gerenciamento de Banco de Dados

SGBDR – Sistema de Gerenciamento de Banco de Dados Relacional

SGBDN – Sistema de Gerenciamento de Banco de Dados na Nuvem

NoSQL – Not Only SQL

BD – Banco de Dados

BDN – Banco de Dados na Nuvem

DHT – Distributed Hash Table (Tabela Hash Distribuída)

DAAS – Database as a Service (Banco de Dados como Serviço)

GFS – Google File System (Systema de Arquivo Google)

AWS - Amazon Web Service (Serviço Web da Amazon)

DFS - Distributed File System (Sistema de Arquivo Distribuído)

CAP - Consistency, Availability and Partition Tolerance (Consistência, Disponibilidade e Tolerância à Partição)

AFD - Accural Failure Detector (Detector de Falha Preciso)

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A quantidade massiva de dados sendo produzidos pela modernização das aplicações e disseminação da Internet, juntamente com a criação de enormes centros de dados (*datacenters*), fizeram surgir um novo paradigma de computação: a computação na nuvem. Vários segmentos deste paradigma vêm sendo evoluídos como forma de serviço, tanto no nível de software como no de hardware. Um desses serviços disponibilizados pela computação na nuvem como plataforma de arquitetura é o Banco de Dados como um Serviço (Database As A Service – DAAS) (Puttini et al 2013). Neste contexto, DAAS são classificados como um Banco de Dados em Nuvem (BDN), representando um desafio e trazendo outras características além daquelas presentes em Sistemas de Banco de Dados Convencionais. Com o intuito de desfrutar dos benefícios da nuvem, um BDN precisa ter arquitetura distribuída, o que limita ao sistema cumprir uma das três propriedades de um Sistema de Gerenciamento de Banco de Dados (SGBD): Disponibilidade, Consistência e Tolerância à partição de rede. Tais conceitos serão detalhados nas seções 1.1 e 1.2.

1.1. Contexto da nuvem

Com o intuito de definir o que é Banco de Dados nas Nuvens antes é preciso definir o termo computação nas nuvens. Várias definições do termo existem na literatura como um todo, no entanto a proposta por Puttini e Erl (2013) provê uma definição concisa:

“Cloud Computing é uma forma especializada de computação distribuída que introduz modelos de utilização para prover remotamente recursos escaláveis e medidos. ”

Isto significa que computação na nuvem se refere a um modelo de computação distribuída que faz uso de recursos físicos e virtuais como serviço, sendo medidos e escaláveis, isto é, com capacidade de variar a quantidade de processamento e memória conforme a demanda, através da rede.

Assim, para que um sistema seja considerado um BDN, uma das poucas requisições é que ele forneça, por meio da rede, persistência e recuperação de dados na forma de serviços. Considerando que o sistema está sendo operado em um centro de processamento de dados (*datacenter*) com acesso a inúmeras máquinas, um BDN eficiente é projetado para utilizar esses recursos físicos (Pinheiro, 2013).

1.1.1. Banco de Dados Convencionais

De acordo com Navathe et al. (2005), Sistemas de Banco de Dados são caracterizados por um banco de dados e um software de Sistema de Gerenciamento de Banco de Dados (SGBD). Sendo que o SGBD é uma coleção de programas que permitem ao usuário criar e manter bancos de dados. O SGBD é, em outras palavras, um sistema de software que facilita os processos de definição, construção, manipulação e compartilhamento de bancos de dados entre vários usuários

e aplicações. A definição de banco de dados refere-se ao fato de especificar os tipos de dados, as estruturas e as restrições para os dados serem armazenados em um banco de dados. Sistemas de Gerenciamento de Banco de Dados Convencionais são projetados para operar em apenas uma máquina ou um conjunto de máquinas com localização física próxima umas das outras. Nesse contexto, o SGBD mais popular é o Relacional (SGBDR) (Pinheiro 2013). Ele permite que transações sejam executadas com propriedades de Atomicidade, Consistência, Isolamento e Durabilidade, ainda permitindo restrições complexas sobre os dados. Por um sistema em nuvem ser consistido de uma arquitetura distribuída, este modelo de SGBD convencional não se adequa às restrições da seção anterior por não administrar bem um grande número de máquinas, de modo que ele não aproveita da melhor forma a escalabilidade, uma propriedade fundamental da Computação nas Nuvens.

1.1.2. A Restrição CAP

Para que um BDN opere de forma eficiente, é necessário que mais de uma máquina física esteja funcionando em um ambiente de nuvem. Dessa forma, o teorema CAP (Brewer, 2012) passa a ser importante em relação ao desenvolvimento do BDN.

1.2. Motivação

Aplicações atuais que demandam uma enorme quantidade de dados buscam por formas de manter a disponibilidade do sistema constante. É preferível, em grande parte, que o sistema se mantenha em funcionamento, mesmo podendo conter dados inconsistentes em relação à arquitetura. Desta forma, é imprescindível utilizar novos conceitos e técnicas que conciliem a disponibilidade do sistema com o dado mais recente. Mesmo com dados desatualizados em réplicas, há grande expectativa de definir melhores políticas que ofereçam acesso eficiente à informação.

1.3. Objetivo do trabalho

O objetivo deste trabalho é realizar uma análise comparativa entre os principais sistemas de bancos de dados existentes no mercado em relação à disponibilidade dos dados, visto que certos sistemas devem priorizar esta característica mesmo em relação à consistência dos dados. Fatores como balanceamento de carga e falhas na rede serão analisados. Por fim, uma será apresentada uma descrição com mais detalhes de uma aplicação focando em características de disponibilidade no Cassandra Database[1]. De acordo com (Zhang, 2014), o sistema Cassandra se caracteriza por um sistema flexível e mais eficiente em relação aos bancos de dados na nuvem MongoDB[2] e HBase[3] quando o volume de dados é muito grande. Como exemplos destes sistemas de banco de dados, tem-se o Amazon Dynamo (DeCandia et al., 2007) que oferece

[1] http://cassandra.apache.org/
[2] https://www.mongodb.org/
[3] https://hbase.apache.org/

grande escalabilidade dos dados e baixa latência, e o sistema BigTable[4] que armazena grandes quantidades de dados, utilizados em aplicações como o Google Maps[5].

Como objetivo específico desta monografia, é apropriado realizar um estudo sobre fatores que influenciam a disponibilidade em Sistema de Gerenciamento de Dados em Nuvem (SGBDN) e a motivação de aplicações pela escolha de tais sistemas.

1.4. Organização do Documento

Além deste capítulo, o trabalho está estruturado como segue. O Capítulo 2 traz alguns aspectos sobre *datacenters* e sistemas distribuídos, além de técnicas gerais utilizadas por Sistemas de Banco de Dados (SGBD) nas Nuvens e os critérios de escolha dos sistemas que serão explorados em detalhes nos capítulos seguintes. Os capítulos 3 a 5 descrevem os sistemas BigTable, Dynamo e Cassandra. O Capítulo 5 apresenta casos de uso que indicam o principal uso dessas tecnologias. O Capítulo 6 expõe as conclusões, as novas tecnologias emergentes, limitações e sugestões de trabalhos futuros.

[4] https://cloud.google.com/bigtable/
[5] https://www.google.com.br/maps

2. BANCO DE DADOS NAS NUVENS

Atualmente, aplicações fazem uso cada vez maior de recursos computacionais com armazenamento e processamento eficiente. Dessa forma, foi fundamental o surgimento de um paradigma de sistema de gerenciamento de dados baseado com conceitos de escalabilidade. Segundo Pramod e Martin (2013) o termo NoSQL (Not Only SQL) não é somente aplicado a um certo número de banco de dados não-relacional como Cassandra, MongoDB, e Neo4J, mas também pode requerer o uso de SGDBR. Sistemas de banco de dados relacionais constituem uma ferramenta muito poderosa, sendo complementados por SGBDN que ofereçam suporte a aplicações que requerem *persistência poliglota*, pelo uso de várias tecnologias para gerenciar dados de acordo com a necessidade do negócio.

Em sua grande maioria, um BDN é caracterizando por linguagem de consulta baseada em NoSQL, embora haja sistemas nos quais a linguagem utilizada utiliza conceitos de um banco de dados relacional. Contudo, não há um padrão para linguagem deste tipo, sendo um dos principais fatores que influenciam como um sistema deve obedecer tais restrições de disponibilidade. Dessa forma, muitos sistemas deixam a cargo do usuário definir o nível de consistência a fim de obter ganhos com a disponibilidade.

2.1.1. Definições e nomenclaturas

De acordo com Haerder et al. (1983), independentemente do número de transações sendo realizadas ao mesmo tempo, SGBDR clássicos garantem que cada uma dessas operações possua características ACID:

1. Atomicidade (A): o sistema executa a transação por completa ou simplesmente não a executa. Ou seja, a transação deverá ocorrer de forma atômica sem haver outros efeitos de execução ou com falhas;
2. Consistência (C): neste caso, as restrições no que se refere a regras de integridade no banco de dados são respeitadas;
3. Isolamento (I): demais transações não sofrem o efeito de uma transação não realizada com sucesso; e
4. Durabilidade (D): os efeitos de uma transação realizada com sucesso tornam-se persistente independentemente do estado do banco de dados.

Porém, em se tratando da arquitetura de um SGBDN, a consistência dos dados sofre influências de acordo com a partição da rede, classificando tais sistemas nas características BASE (*Basic Availability, Soft state, Eventual consistency*). Neste caso, para dispor de alta disponibilidade, a consistência pode ser sacrificada com desatualização em um certo número de réplicas do banco de dados, tornando o sistema eventualmente consistente.

2.1.2. Latência

Segundo Coulouris et al. (2013), latência é o tempo decorrido após uma operação de envio ser executada e antes que os dados comecem a chegar a seu destino. Em sistemas distribuídos, é

comum referenciar a latência da rede que forma uma parte da latência de todo o processo. A tolerância à falha em partição da rede tem relação direta com a latência do sistema. É notório distinguir quando um sistema distribuído falha ou está inatingível de um sistema que está demorando a dar resposta.

2.2. Aspectos e Técnicas Gerais de SGBD nas Nuvens

Informação sensível sendo gravada em ambiente na nuvem e em *datacenters* precisam de serviços de banco de dados em nuvem. Mas ainda assim, um emergente mercado de serviços *de cloud database* e outras ferramentas está crescendo rapidamente. Alguns provedores trabalham diretamente com modelos relacionais, sistemas de banco de dados SQL ou NoSQL, enquanto outros são focados em banco de dados *open source*, como Firebird[6] ou PostgreSQL[7] (Brandon, 2014).

Para prover os serviços aos usuários de forma persistente e com recuperação de dados, empresas do ramo de informática trabalham com uma variedade de serviços, incluindo banco de dados relacional e em NoSQL. A exemplo disto pode-se citar o Amazon Web Services (AWS)[8] que utiliza instâncias do Oracle[9], MySQL[10] ou SQL Server[11] como SGBD relacionais. Ainda, pode-se citar o uso do SimpleDB[12] como sistema de banco de dados com esquema mais simples para pouca carga de trabalho, e do lado NoSQL, o Amazon Dynamo[13] que automaticamente replica cargas de trabalho por meio de pelo menos três zonas de disponibilidade (DeCandia et al., 2007).

2.2.1. Modelo de Dados e Sistema de Persistência

O armazenamento e processamento de consultas são pontos críticos no contexto de gestão de dados na nuvem (Armbrust et al., 2009). Existem diversas abordagens para gerenciar dados na nuvem e cada sistema utiliza uma abordagem específica para persistir os dados. Dentre estas abordagens, *frameworks*, sistemas de arquivos e propostas são destacados para armazenar e processar tais dados.

No contexto de dados de sistemas NoSQL, a modelagem para prover armazenamento e processamento dos dados é classificada tanto como a abordagem multidimensional quanto no esquema *key-value*. Na abordagem multidimensional, os dados estão presentes em tabelas

[6] http://www.postgresql.org/
[7] http://www.firebirdsql.org/
[8] https://aws.amazon.com/pt/rds/
[9] http://www.oracle.com/br/database/overview/index.html
[10] https://www.mysql.com/
[11] http://www.microsoft.com/pt-br/server-cloud/products/sql-server/
[12] https://aws.amazon.com/pt/simpledb/
[13] https://aws.amazon.com/pt/documentation/dynamodb/

acessadas a partir de chaves nas colunas, sendo os dados posteriormente agrupados. Já na abordagem *key-value*, uma estrutura de chave-valor é utilizada em uma Distributed Hash Table (DHT) (DeCandia et al., 2007). A chave e os valores são armazenados em estruturas desnormalizadas, facilitando a distribuição dos dados entre os nós do sistema. Cada chave é utilizada para realizar o acesso ao valor, que pode ser de qualquer agregado de dados, ficando a cargo da aplicação definir a melhor forma utilização. Para acesso aos dados, a API oferece operações simples tais como put (*key, value*) e get(*key*). Neste caso, o modelo de dados faz o acesso aos dados por meio da estrutura DHT com alto desempenho por realizar a busca por meio de índices. O custo dessas operações de consultas é reduzido representando no máximo o tamanho da estrutura em sistemas de nós distribuídos.

Para persistir os dados, os SGBDN utilizam um sistema de arquivos distribuídos (DFS) com a propriedade de permitir o acesso a arquivos armazenados em servidores remotos. Contem implementação de controle de concorrência (Carmem, 2011), sendo possível fazer replicação para maior disponibilidade dos dados e tolerância a falhas.

2.2.2. Particionamento dos Dados

Grandes benefícios são obtidos por um sistema distribuído por meio da técnica de particionamento dos dados, como aumento da escalabilidade e alto desempenho do processamento. Assim como a maioria dos sistemas NoSQL, os SBDN analisados utilizam a abordagem de particionamento dos dados de alguma forma na execução ou dependem de sistemas externos que realizam tal técnica.

No momento em que o Sistema de Banco de Dados na Nuvem utiliza de partição, cada nó deve ser responsável pelo armazenamento dos dados relacionados com a identificação do valor pelo nó. Há basicamente duas técnicas de particionamento de dados (ASP NET, 2015):

1. *Particionamento horizontal* - esta técnica, também chamada de *sharding*, se assemelha à separação de tabela pelas linhas, um conjunto de linhas deve ir para um nó de armazenamento de dados, enquanto o outro conjunto deve ir para um diferente servidor de dados; e
2. *Particionamento vertical* - este tipo de particionamento de dados ocorre em uma tabela pela separação dos dados por meio das colunas, sendo que um conjunto de colunas é separado em um servidor de dados enquanto outro conjunto de colunas é separado em um nó diferente no sistema.

Também há as abordagens de partição relacionadas levando em consideração a característica de localização física dos dados. Duas formas comuns de realizar tal técnica são (Pinheiro, 2013):

1. *Partição por intervalo* – o processo consiste em dividir o espaço das chaves em intervalos, e atribuir a cada nó um intervalo distinto; e
2. *Partição por Hash* – Semelhante à partição por intervalo, porém a chave é gerada por meio de uma função *hash*.

Um dos principais fatores para o particionamento está no fato de estabelecer o balanceamento da carga de trabalho entre os nós no sistema distribuído. O DynamoDB, por exemplo, distribui o espaço de chaves entre os servidores e assume que o número de requisições será em média na mesma proporção.

2.2.3. Replicação

A replicação permite criar outra instância de banco de dados que pode ser usada para dimensionamento de carga de trabalho ou assegurar disponibilidade do banco de dados em caso de falha da instância. Refere-se, então, à manutenção de cópias idênticas de dados em locais diferentes. Em relação ao tipo de replicação esta pode ser classificada como síncrona ou assíncrona.

Na replicação síncrona, caso o banco de dados sofra alteração, a modificação será propagada e aplicada em outros servidores de dados em um segundo momento dentro de uma transação à parte, certo tempo depois. Já na replicação assíncrona, todas as cópias ou replicações de dados serão realizadas no instante da sincronização e consistência. Quando uma alteração no banco for executada, as outras réplicas serão imediatamente atualizadas, neste caso ocorre uma atualização ansiosa (Freitas et al., 2015).

Em relação ao teorema CAP, a abordagem do tipo de replicação pode prover um tipo de consistência forte ou eventual de acordo com a atualização, sendo dessa forma decisivo para o tipo de disponibilidade que o sistema deve atuar. Por exemplo, o sistema DynamoDB preza pela alta disponibilidade dos dados com a característica de ter consistência eventual.

2.2.4. Versionamento

Para manter versões de dados em relação ao tipo de escrita em controle de versões, é preciso definir uma estrutura que possa gerenciar as últimas atualizações em relação aos dados presentes nos nós do sistema distribuído. Tal técnica pode ser relevante ou não quanto às características CAP. Dessa forma, o termo será utilizado para definir métricas de versões distintas do mesmo dado quanto à atualização nos nós distribuídos. Dois exemplos de versionamento (De Candia et al., 2007; Lakshman, 2009; Pinheiro, 2013):

1. Versionamento desejável e pouco impacto à CAP – Versões antigas dos dados são mantidas em sistemas de tabelas multidimensionais propositalmente, a fim de serem usadas pelos clientes. Quando o cliente acessa o dado, a versão mais recente é acessada a não ser que seja explicitada uma versão mais antiga do mesmo; e
2. Versionamento indesejável e relevante à CAP – Sistemas com garantia de consistência fraca podem estar em um estado de consistência eventual, ou seja, inconsistente num certo tempo, e ter versões distintas do dado espalhados por diversos nós, precisando aplicar políticas complexas para decidir qual usar, ou entregar ao cliente qualquer uma disponível.

2.2.5. Detecção e Recuperação de Falhas

Uma característica importante para um sistema de banco de dados na nuvem é prover mecanismo de identificar e recuperar as falhas ocorridas. Cada sistema segue uma série de abordagens, mas em geral a técnica de *Master-Slave* representa aquela mais adotada, na qual há dois tipos de nós:

Coordenado – Neste caso os nós do sistema são coordenados por uma entidade coordenadora, sendo responsável por detectar falhas e recuperá-las. Contudo, o próprio coordenador pode falhar. Desta forma, o sistema utiliza uma máquina de backup para atribuir o papel de novo coordenador dos nós restantes automaticamente; e

Não coordenado – existe autonomia em cada nó, o qual identifica quais são aqueles que estão disponíveis ou não no cenário atual do sistema.

É importante notar que a escolha do novo nó coordenador depende de características voltadas à arquitetura do sistema, como prioridade de escolha baseado na memória e processamento disponível nas máquinas mais próximas em *datacenters* ou em relação à política de tráfego na rede.

2.2.6. Características Não Abordadas

Existem características que se tornam importantes para sistemas de banco de dados na nuvem, indicando grau de afinidade em certas aplicações, porém fogem do escopo da abordagem deste trabalho e não serão abordadas profundamente.

A técnica de *Map-Reduce*, por exemplo é uma das responsáveis pelo sucesso do sistema BigTable (Fay Chang et al., 2006), permitindo o acesso a grandes quantidades de dados de forma eficiente sendo agrupadas por funções. No entanto, essa é uma técnica complementar ao sistema e não tem efeito sobre as características exploradas neste trabalho.

2.2.7. Foco de cada sistema

Em relação às características CAP, cada SGBD distribuído pode ser classificado de acordo com o tipo de enfoque. O Quadro 2.1 mostra uma classificação de alguns sistemas quanto a suas características predominantes. Nele, o ‘x’ representa que o sistema se enquadra nessa categoria, enquanto que o ‘–‘ indica que o sistema não está classificado em tal categoria.

Quadro 2.1 – Classificação de alguns sistemas de banco de dados em nuvem quanto a CAP

Sistema	Nativo	AP ou BASE	ACID ou CA	CP	Variável em tempo de execução
Bigtable	x	-	-	x	-
Amazon DynamoDB	x	x	-	-	-
CASSANDRA	-	-	-	-	x
Amazon RDS	-	-	x	-	-
CouchDB	-	x	-	-	-
SimpleDB	x	x	-	-	-
PNUTS	-	-	-	-	x
Neo4j	-	-	x	-	-
HBASE	x				

Fonte: Adaptado de Pinheiro (2013)

Sistemas nativos são aqueles que foram projetados inicialmente para trabalhar na nuvem, enquanto os sistemas não nativos inicialmente não foram projetados para nuvem, porém podem trabalhar nesse contexto.

2.3. Considerações Finais

O presente capítulo discorreu sobre alguns conceitos utilizados em banco de dados em nuvem, além de mostrar uma variedade de possibilidades nas quais diferentes níveis de CAP podem ser atingidos de acordo com o foco de cada BDN. Uma visão geral sobre a abordagem NoSQL evidenciou que a escolha de replicação, versionamento e partição de rede tolerante a falhas vai depender do nível de consistência que a aplicação deseja obter. A disponibilidade deve ser afetada considerando a consistência a ser tratada e a forma de recuperação de falhas.

Nos próximos capítulos serão apresentados aspectos e técnicas relevantes em relação a alguns sistemas. Será dada maior prioridade para os sistemas BigTable, Dynamo e Cassandra, já que estes sistemas apresentam características complementares entre si e implementações distintas, além de abrir mão de aspectos diferentes em características do CAP.

Dynamo e BigTable são descritos mais adiante, pois suas características influenciaram o conceito e implementação de outros sistemas de banco de dados distribuídos, como por exemplo Cassandra e HBase.

Cassandra será por fim abordado por conter propriedades distintas dos sistemas de bancos de dados citados anteriormente por dispor de certa flexibilidade do modelo de dados nas colunas, com alta escalabilidade e agilidade de processamento *in memory*.

3. BIGTABLE

Este capítulo fornece uma visão geral sobre o sistema BigTable. Primeiramente descreve-se o motivo pelo qual o sistema foi criado e aspectos de sua construção. Em seguida, segue uma discussão das principais características do BigTable que influenciaram as características CAP e por último, uma análise será realizada com uma comparação entre as técnicas descritas até o presente tópico.

3.1. Arquitetura do sistema

BigTable pode ser considerado como um sistema distribuído com o intuito de gerenciar dados armazenados em larga escala na faixa de petabytes, espalhados em servidores pelo mundo. A implementação do sistema, de acordo com (Chang et al., 2006), é composta por três maiores componentes: uma biblioteca incorporada em cada cliente, um servidor *master* e vários servidores *tablets* (*tabletservers*). O servidor *master* é responsável pelo gerenciamento dos servidores *tablet*, com o balanceamento de carga, a adição de servidores *tablets* e o *garbage collection* de arquivos no Google File Sytem - GFS (Sanjay et al., 2003), usando o sistema Chubby (Burrows, 2015).

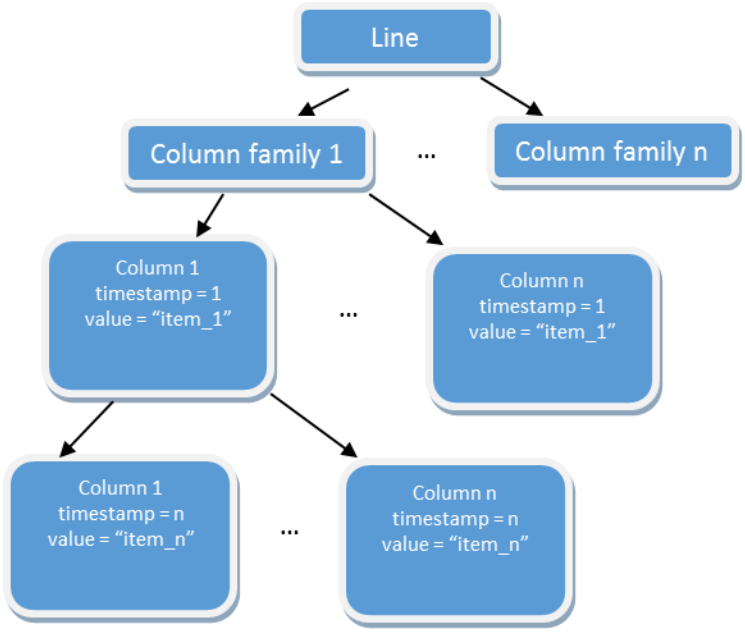
3.1.1. Modelo de Dados

A arquitetura do sistema BigTable é organizada em uma tabela multidimensional. Cada entrada da tabela é identificada por uma família de colunas, uma chave (que indica a linha), um qualificador da coluna e um *timestamp*. Uma família contém uma lista de qualificadores. Todas as linhas de uma tabela possuem as mesmas famílias de coluna, porém cada coluna é flexível. Dentro de uma posição, referente a uma linha específica e a uma coluna, podem existir várias versões distintas do item, cada uma com um *timestamp* associado.

As tabelas são otimizadas pelo Sistema de Arquivos da Google sendo quebradas por múltiplos pedaços chamados *tablets,* que são segmentos da tabela divididos ao longo de uma linha escolhida tendo por volta de 200 MB de tamanho.

Cada chave é representada por uma cadeia de caracteres, sendo ordenadas alfabeticamente na tabela. A Figura 3.1 ilustra uma representação em hierarquia da estrutura de uma tabela do sistema BigTable, com os componentes associados.

Figura 3.1 – Representação em hierarquia de uma tabela



Fonte: O Autor

3.1.2. Google File System (GFS)

No intuito de gerenciar uma enorme quantidade de dados para ser retida, o Sistema de Arquivo da Google (GFS – Google File System) (Sanjay et al 2003) é um sistema de armazenamento distribuído proposto para prover maior confiabilidade e eficiência no acesso aos dados usando enormes clusters.

Em sua arquitetura, o GFS é formado por um servidor mestre (*master*) e vários servidores *chuncks* (*chunckserver*). Cada arquivo é subdividido em vários pedaços (*chuncks*), sendo atribuído a cada pedaço um identificador. A localização física de cada *chunck* é mapeada pelo servidor mestre para a aplicação, assim como qual arquivo cada *chunk* constitui. O problema da tolerância a falhas é atenuado pela redundância dos dados nos arquivos (*chunk files*), bem como

pela eleição de novo *shadowserver* (servidor secundário) para substituir o servidor mestre caso este venha a falhar.

O servidor mestre recebe constantemente uma mensagem de cada servidor *chunck* indicando que este está operando de forma coerente. Caso esta mensagem não tenha sido recebida após certo período, ou caso o servidor não responda a uma mensagem, este considera que houve falha no processo de envio e todas as cópias dos arquivos presentes naquele servidor são perdidas. O servidor mestre fica responsável por criar uma nova réplica se o número de réplicas cair abaixo de certo limite.

Para executar uma operação de escrita o cliente precisa consultar o servidor mestre. A escrita concorrente não é permitida pelo servidor mestre em um mesmo pedaço de arquivo. Da mesma forma, o servidor bloqueia um acesso de leitura enquanto houver uma escrita a ser realizada. O servidor que contém o pedaço de arquivo armazenado recebe a requisição do cliente por meio do servidor mestre caso esta tenha sido atendida. Então o *master* propaga a informação para ser gravada nas demais réplicas, finalizando a execução do processo caso todas as réplicas tenham confirmado o recebimento da requisição (Sanjay et al 2003).

3.1.3. Chubby

De acordo com Chang et al. (2006), Chubby é um sistema distribuído e persistente de alta disponibilidade que consiste em cinco réplicas ativas, umas das quais é eleita para ser *master* e constantemente servir requisições. O Chubby utiliza um algoritmo da família Paxos (Chandra et al., 2007) para manter suas réplicas persistentes num mesmo ambiente onde as falhas ocorram. Este algoritmo é parte fundamental do funcionamento do BigTable, por conferir um mecanismo de sincronização dos dados em réplicas e consiste em três fases, segundo Chandra et al. (2007), que podem se repetir:

1. Uma réplica é eleita para ser a coordenadora;
2. O nó coordenador seleciona um valor e envia uma mensagem *accept* para todas as outras réplicas em *broadcast.* As outras réplicas podem aceitar ou rejeitar a mensagem; e
3. Caso a maioria das réplicas aceite a mensagem do coordenador, um consenso foi criado, e o coordenador envia uma mensagem de *commit* em *broadcast* para notificar todas as réplicas.

3.1.4. SSTable

O formato de arquivo Google SSTable é usado internamente para armazenar dados no Bigtable. Formado por uma sequência de blocos em torno de 64KB, porém configurável, o SSTable contém blocos de índice que podem ser completamente mapeados na memória, melhorando a busca e acesso sem “tocar” no disco (Chang et al., 2006). A grande importância do sistema neste formato está na representação dos dados que já são inseridos com índices, facilitando a eficiência com que as consultas são realizadas.

3.2.Lidando com CAP

Algumas técnicas presentes no sistema que afetam diretamente as características CAP serão descritas nas seções 3.2.1 e 3.2.2, como a replicação e particionamento que afetam diretamente a disponibilidade e o versionamento que lida com o nível de consistência do banco de dados.

3.2.1. Replica e Versionamento

De acordo com o contexto de um *datacenter*, o BigTable não utiliza nenhuma técnica de réplica diretamente, embora a técnica seja essencial no GFS e no Chubby (Pinheiro, 2013). O BigTable pode ser configurado para conter várias réplicas em outros *datacenters*, aumentando a disponibilidade dos dados. Porém, como as réplicas são atualizadas aos poucos, numa abordagem assíncrona, o sistema então passa a ter uma consistência eventual.

Cada posição na tabela (linha, família de coluna, identificador da coluna) pode possuir várias versões do dado, com um *timestamp* diferente de acordo com a Figura 2.1. Como o sistema bloqueia a linha inteira, caso haja um processo de escrita em alguma coluna, não haverá leitura em outro campo da família de colunas, de forma que inconsistências não são criadas. No entanto, existe a possibilidade de o cliente fornecer o *timestamp* que deve estar associado ao dado. Neste caso, portanto, o sistema não garante que não ocorra conflito com diferentes versões do mesmo dado.

Dois critérios são utilizados pelo sistema para excluir uma versão do dado (Chang et al 2006):

1. O BigTable descarta a versão mais antiga do dado caso a quantidade de versões exceda um limite preestabelecido pelo cliente; ou
2. Versões mais antigas dos dados são descartadas em relação a certo tempo fornecido pelo usuário.

3.2.2. Particionamento dos dados

As tabelas são particionadas com replicação horizontal considerando a chave, mas também pode haver particionamento vertical de acordo com a família de colunas.

O sistema utiliza dois tipos de tabelas que são tratadas de formas especiais. São diferenciadas as tabelas de metadados que contém as informações das partições, e uma tabela raiz, que guarda a localização das tabelas de metadado. A tabela raiz não é particionada, para evitar excesso de consulta e diminuir a latência ao descobrir posições nas tabelas de metadados.

3.2.3. Vantagens e desvantagens

Alguns erros comuns presenciados pela aplicação Google Maps[14] em relação à persistência dos dados na aba Favoritos[15]. Por vezes esses dados não estão presentes ao iniciar o aplicativo, mostrando erros ao tentar carregar arquivos salvos na guia ou ao tentar remover um evento favorito. Outro grande aspecto, refere-se à forma de gerenciamento do sistema BigTable ficando a cargo da aplicação escolher o tipo de atualização a ser realizada. Caso se trate de uma atualização *single row*, o sistema provê alta consistência dos dados. No entanto, caso a transação feita seja de atualização em *multiple row*, a consistência configurável é relaxada, ou seja, os dados contêm consistência eventual (Hofmann e Quora, 2015).

Os dados geográficos referentes ao esquema do Google Maps são armazenados em um formato XML[16] conhecido como KML (Keyhole Markup Language)[17], no qual os dados vetoriais e imagens são recebidos em diversos formatos e em resoluções de diversas fontes. Na tabela dos dados geográficos, as linhas representam os locais geográficos em particular e as colunas representam diferentes aspectos de localização e imagens brutas, organizados em famílias de coluna. Portanto, pelo fato de constantemente haver atualização das múltiplas linhas que representam as imagens, a atualização dos dados geográficos, em grande parte, ocorre lentamente. Contudo, aqueles dados que definem novas ruas, ruas alteradas, e outra infraestrutura se tornam disponíveis o tempo todo (Coulouris et al., 2013).

3.3. Considerações Finais

O modelo de dados do BigTable é baseado em *key-value* e por mante enormes quantidades de dados, esbarra na limitação de resolver conflitos de escrita por parte do usuário. O sistema utiliza um bloqueio nessas operações de escrita com o GFS. Fica a cargo do algoritmo Paxos resolver conflitos de versão dos dados, sem evidenciar essas distinções para o BigTable. Tal técnica torna o sistema pouco flexível às características CAP, uma vez que torna a consistência dos dados forte para dados processados em paralelo. Já em relação ao particionamento dos dados, o BigTable considera o espaço de chaves distribuído em intervalo de acordo com a localidade das réplicas que armazenam estes dados. Tal fato é utilizado para diminuir a latência ao otimizar as consultas, mas tem efeito negativo na disponibilidade pois um nó especializado é responsável por armazenar os valores de cada intervalo da partição.

O próximo capítulo aborda os principais conceitos e técnicas do modelo de dados presentes no Dynamo e aspectos que afetam as características CAP, além de fornecer exemplo de aplicação que necessita de alta disponibilidade e mecanismos usados para prover este serviço.

[14] https://www.google.com.br/maps
[15] https://productforums.google.com/forum/#!msg/maps-pt/40Qmgp-vMpg/s6DaZxM9aZIJ
[16] http://www.tecmundo.com.br/programacao/1762-o-que-e-xml-.htm
[17] https://developers.google.com/kml/documentation/kml_tut?hl=pt-br

4. DYNAMO

Com o objetivo de ser extremamente escalável, o sistema Dynamo (DeCandia et al., 2009), desenvolvido pela Amazon[18], opera com tempo de resposta curto. Uma característica fundamental no seu funcionamento está presente na alta disponibilidade dos dados em relação às operações de escrita. Os dados do sistema, no entanto, possuem consistência eventual.

4.1. Arquitetura do sistema

O funcionamento do Dynamo em relação à arquitetura consiste na explicação do modelo de dados e na utilização de nós virtuais em servidores.

4.1.1. Modelo de dados

O sistema faz uso da modelagem *key-value*, na qual cada tabela armazena dados, os quais possuem apenas os campos chave e valor. A tabela com esta modelagem é indexada e cada valor refere-se a um objeto que pode ser qualquer agregado, ficando a cargo da aplicação tratar de forma conveniente cada informação. O particionamento ocorre por meio de uma técnica chamada *hash* consistente (*consistenthashing*) presente em sistemas distribuídos (Karger et al., 1997). Além disso, cada registro é atrelado a um vetor lógico de versionamento (*vectorclock*) (Baldoni et al., 2002), que será detalhado nas seções seguintes.

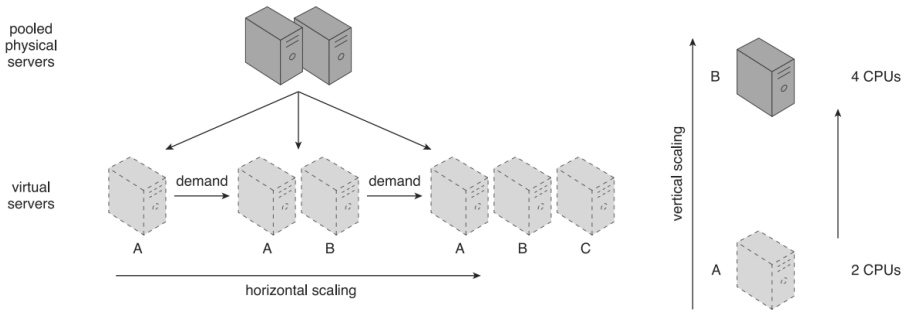
4.1.2. Virtualização

O conceito de virtualização é largamente utilizado em computação na nuvem por prover replicação dos dados de forma mais segura. No caso do sistema Dynamo, tal técnica é enfatizada tanto por trabalhar com a heterogeneidade dos dados entre componentes físicos do sistema, quanto lidar com algumas características relevantes a CAP quando consideradas em conjunto com as técnicas de replicação e versionamento.

Um servidor de armazenamento de dados é considerado um nó físico e pode conter vários nós virtuais dependendo de suas características físicas, como capacidade de armazenamento e taxa de transferência. Esta técnica de virtualização em servidores permite aos sistemas distribuídos aumentarem recursos em termos de escalabilidade horizontal ou vertical. Ambas são tipos de alocação de maiores recursos computacionais. A Figura 4.1 mostra esses dois tipos de escalabilidade. No primeiro caso, os componentes são alocados considerando a demanda de vários recursos do mesmo tipo. Enquanto que na escalabilidade vertical, não há mais adição de recursos computacionais, apenas a agregação de vários recursos em um só nó físico (Puttini et al., 2013).

[18] http://www.amazon.com/

Figura 4.1 – Escalabilidade horizontal e vertical



Fonte: (Puttini et al., 2013)

A tolerância à partição em falhas é garantida por esse tipo de técnica na maioria dos sistemas distribuídos (Puttini et al., 2013).

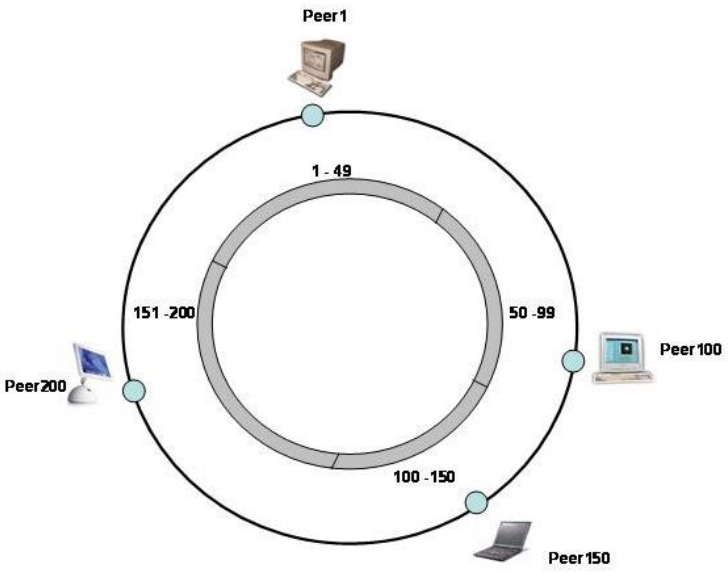
4.2.Lidando com CAP

Serão analisados com mais detalhes as técnicas utilizadas pelo sistema. O modo como o Dynamo utiliza a replicação de dados e o sistema de versionamento baseado em *vectorclocks* são as duas técnicas mais importantes quanto à influência nas características CAP (DeCandia et al., 2007).

4.2.1. Particionamento dos Dados

A técnica usada pelo Dynamo para particionar os dados refere-se à Tabela Hash Distribuída (DHT – Distributed Hash Table) ou *hash* consistente sobre as chaves da entrada. Os valores de saída dessa função podem ser interpretados como um espaço circular ou anel. Cada nó recebe uma numeração na tabela circular de forma aleatória. Quando uma requisição de buscar um dado chega a um nó específico, este utiliza a função *hash* para saber qual o nó na tabela mais próximo ao dado. A função continua sendo aplicada, encaminhando a requisição para outros nós até não ultrapassar o índice do *peer* procurado em sentido horário. Por fim, o nó é encontrado no espaço presente dedicado à sua responsabilidade. A Figura 4.2 mostra a representação de um esquema de DHT, onde cada peer tem a responsabilidade de receber as requisições dos 50 *peers* anteriores.

Figura 4.2 – Representação de uma Distributed Hash Table, na qual cada nó é responsável pelos 50 anteriores.



Fonte: (Moreira, 2015)

Esta abordagem traz as seguintes vantagens (DeCandia et al., 2007):

- A adição ou remoção de um nó físico afeta apenas os nós nos quais seus nós virtuais antecedem e sucedem;

- A carga é distribuída uniformemente entre os demais nós, caso um nó não esteja disponível por algum motivo; e

- Com o intuito de um melhor balanceamento, quando um nó se torna disponível novamente ou o sistema adiciona um novo nó, este passa a receber uma carga de requisições equivalentes a dos outros nós.

4.2.2. Versionamento dos dados

Para garantir alta disponibilidade, o sistema lida com a consistência eventual presente nos nós distribuídos. Por aplicar replicação assíncrona, é possível que o mesmo objeto tenha múltiplas versões distintas num dado instante. Entretanto, em certos cenários de falha as atualizações podem não chegar a todas as réplicas durante certo tempo (DeCandia et al., 2007). Para lidar

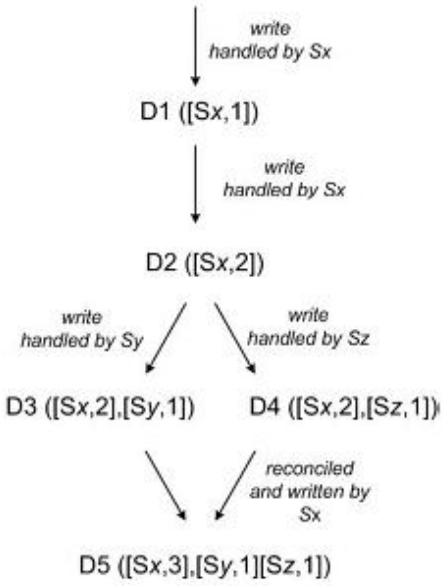
com isso, o Dynamo considera o resultado de toda a operação de escrita como uma nova versão imutável do dado.

A cada nova requisição de leitura, o controle das versões determina qual valor deve ser retornado para evitar conflitos. Na maioria das vezes, versões mais atuais sobrescrevem versões antigas dos dados, ficando a cargo do sistema determinar a versão a ser utilizada no momento (processo conhecido como reconciliação sintática). Contudo, devido a falhas e atualizações, diferentes versões dos dados podem conflitar entre si, fazendo o sistema tornar-se incapaz de decidir qual versão utilizar. Desta forma, o cliente recebe todas as versões atuais válidas do objeto. Fica a cargo do usuário executar uma operação de escrita subsequente mesclando as versões para resolver o conflito (operação conhecida como *collapse).* Esse processo é chamado de reconciliação semântica. O sistema utiliza uma estrutura conhecida como *vectorclock* para auxiliar no controle de versões dos dados.

Como mencionado anteriormente na Seção 4.3, cada entrada num dado possui um tipo de *vectorclock* associado ao mesmo. Esta estrutura representa uma lista de pares contendo um nó e um contador de modificações realizadas nos dados. Os nós determinam a divergência de duas versões de um objeto ou uma ordenação casual entre versões. Quando o sistema realiza uma operação de escrita, o nó que efetuou a operação altera seu *vectorclock* e o propaga para os demais. Dessa forma o Dynamo identifica se uma versão é mais atual que outra e se ela pode sobrescrever mais antiga. Abaixo o comportamento do sistema diante de tais solicitações para ilustrar esse procedimento:

1. O usuário realiza uma nova requisição de escrita para um dado. O nó (SX) que recebe a escrita para esta chave, aumenta o valor do contador e usa-o para criar um *vectorclock* do dado. O sistema agora tem o objeto D1 e seu clock associado [(SX, 1)];
2. Quando outro usuário realiza uma operação de atualização do dado com o mesmo servidor, logo após D1 ser efetuada, o sistema incrementa o contador no *vectorclock* passando a ter uma nova versão D2;
3. Seja um cliente que realiza uma requisição de leitura a partir de D2 e então tenta atualizar com um servidor Sy, e outro usuário requisita uma operação de escrita por meio do servidor Sz, por exemplo a adição de um novo item no carrinho de compras *shipping cart*. Duas novas versões serão criadas D3 e D4 pela modificação do objeto na última versão.
4. Uma requisição de leitura deverá considerar, portanto, que o contexto da informação persiste nas versões [(Sx, 2), (Sy, 1) e (Sz, 1)]. Caso um cliente opte por fazer uma atualização do dado, esta deve ser reconciliada tomando como base ambos neste último contexto, pois há mudança em D3 e em D4 que não refletem uma na outra. Dessa forma, é necessário que o sistema realize a operação de reconciliação semântica dos dados, incrementando o contador onde a requisição foi realizada (no caso em Sx) e obtendo o novo contexto [(Sx, 3), (Sy, 1) e (Sz, 1)].

Figura 4.3 – Versionamento Dynamo



Fonte: (DeCandia et al., 2007)

A Figura 4.3 mostra a hierarquia das versões dos dados em *vectorclocks* armazenados pelos nós no cenário mencionado acima.

4.2.3. Amazon Shopping Cart

Por prover consistência eventual, consistindo de atualização assíncrona nas réplicas. Uma operação de escrita *put()* pode retornar ao nó que chamou antes que a atualização ocorra em todos as outas réplicas, o que pode resultar em cenários com sucessivas leituras de dados desatualizados em operações *get().* Segundo (DeCandia 2007), a aplicação *Amazon Shopping Cart* pode tolerar tais inconsistências dos dados operando sobre certas condições. Por exemplo, a aplicação requer que a operação *"Adicionar ao Carrinho"* nunca se perca ou seja rejeitada. Mesmo com o estado mais recente do carrinho indisponível, se o usuário realiza uma mudança para um estado antigo do carrinho essa mudança é ainda significativa e deve ser preservada. Mas ao mesmo tempo ele não pode substituir o estado atual indisponível do carrinho que pode conter mudanças a serem preservadas. Quando um cliente quer adicionar ou remover um item ao carrinho e a última versão não está disponível, o item é adicionado ou removido e a versão mais antiga e versões divergentes são reconciliadas posteriormente.

4.2.4. Sucesso nas operações e fator de replicação

Assim que uma requisição de escrita ou leitura é executada em um nó X vindo do cliente, este propaga a operação inicialmente para N réplicas sadias (que X possa alcançar, ou seja, sem que haja falhas). A operação é efetivada se pelo menos K réplicas realizem com sucesso a operação. O valor de K é dado pelo número de escritas W e leituras R.

No sistema Dynamo, pode haver diferentes valores para N, R e W, conquanto que o fator de replicação R+W > N seja obedecido, assunto este tratado com mais detalhe na Seção 5.3.1. Esses valores impactam de forma considerável as características CAP. A tolerância a falhas é garantida por maior valor de N, enquanto maiores valores de R e W garantem maior disponibilidade ou consistência dos dados.

4.3. Considerações Finais

Uma das principais limitações de esquema do Dynamo assim como o Cassandra, consiste no gerenciamento de dados com índices secundários, enfatiza Gross, (2013). A indexação ocorre por meio do valor da chave primária, sendo preciso planejar bem o modo de mapear a função *hash* para cada chave primária. Na atual implementação do Dynamo (Limits in DynamoDB, Amazon 2015), o número de índices secundários locais e globais é limitado por tabela, sendo preciso mais custo para realizar busca por índice primário. Tal processo torna o número de escritas mais caro.

Em termos de CAP, o sistema permite que o usuário defina o número de unidades de escrita (W) e unidades de leitura (R), onde:

1. Uma unidade de leitura: com o mínimo de duas consistências eventuais ou uma consistência forte por segundo para itens maiores que 4KB em tamanho; e
2. Uma unidade de leitura: representa uma escrita por segundo para itens com capacidade de armazenamento acima de 1KB.

5. CASSANDRA

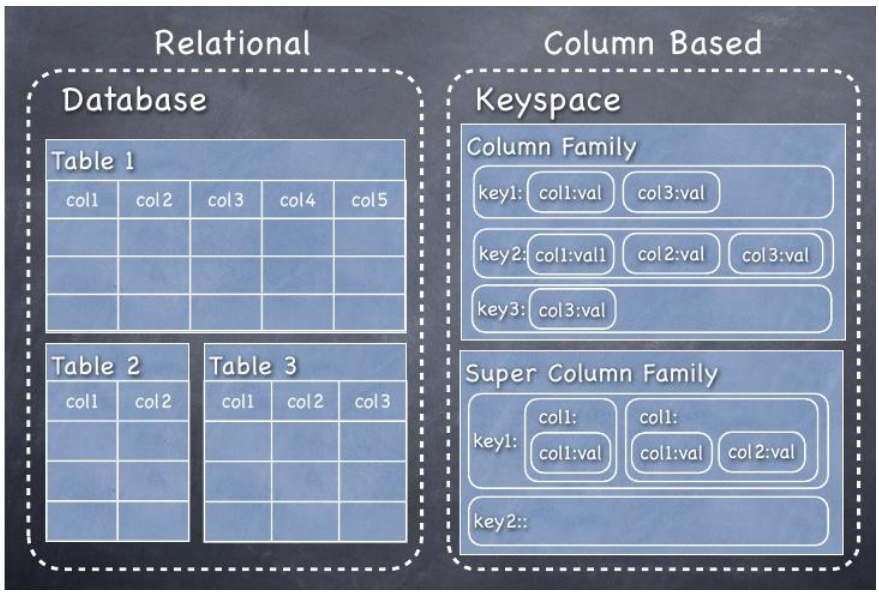
Por tratar de um sistema com modelagem flexível e armazenamento real-time, o sistema Cassandra tem a temática de colocar a cargo do usuário a necessidade das características CAP. É responsabilidade do cliente definir o nível de consistência a ser usado na aplicação, o qual pode ser ajustável dependendo da intenção desejada.

5.1. Arquitetura do sistema

Baseado em armazenamento de família de colunas, o sistema possui chaves mapeadas para valores e estes valores são armazenados em várias famílias de colunas, cada uma sendo um mapeamento de dados. Outros exemplos de sistemas desse tipo são o HBase e o SimpleDB (Gutierrez, 2015).

Com alto poder de escalabilidade, o sistema Cassandra usa o paradigma de sistemas distribuídos *peer-to-peer* (P2P) com réplicas de leitura e escrita. Nesta abordagem, um *Keyspace* representa um banco de dados, as *Column Family* representam as tabelas e os valores (col1:val) de cada key representam uma coluna em um Sistema de Banco de Dados Relacional (SGBDR). Dessa forma, é possível ter uma quantidade variável e dinâmica de colunas para a mesma linha, que no caso do *Column-Family* é uma mesma *key*. As *Super Column Family* são um agregado de *Column Family*. São colunas que representam um mapa de várias colunas (Figura 5.1).

Figura 5.1 – Escalabilidade horizontal e vertical



Fonte: (Gutierrez, 2015)

No sistema, a coluna representa a unidade básica de armazenamento. Uma coluna consiste de um par nome-valor (name-value), onde a key está representada por um nome. Cada um destes pares é uma única coluna com um valor *timestamp* para versionamento dos dados, além de lidar com escalonamento, conflito de escrita, entre outros.

5.2.Lidando com CAP

A seguir serão descritas as técnicas utilizadas pelo sistema Cassandra para lidar com as restrições CAP. A replicação que utiliza a equação de quórum baseado no número de nós escritores e leitores, assim como o uso da técnica de *hash consistente* para particionar os dados é fundamental em se tratando de aspectos da disponibilidade.

5.2.1. Replicação

Pelo fato de trabalhar com memória *in-place*, a operação de escrita ocorre com sucesso após duas fases. A primeira consiste em gravar os dados em um *log* e na segunda acontece uma gravação em estrutura de memória chamada *memtable*. As escritas são agrupadas em memória e periodicamente repassadas para estruturas conhecidas como *SSTable*. Caso ocorra alguma alteração de dados, uma nova *SSTable* é criada e as *SSTable* não usadas voltam para compactação. O processo chamado compactação do Cassandra periodicamente consolida as *SSTables*, descartando dados obsoletos e *tombstones* (indicadores de que os dados foram excluídos).

As configurações do sistema Cassandra são ajustáveis no nível de consistência, de acordo com o *ConsistencyLevel* (DataStax, 2015)[19]. Os níveis de consistência consistem de uma série de intervalos onde cada nível requer restrições de escrita em uma certa quantidade de nós no cluster. Os níveis ONE e ALL se referem mais extremos de valores de consistência enquanto o QUORUM estabelece restrição de consistência baseada em um quórum de nós no cluster.

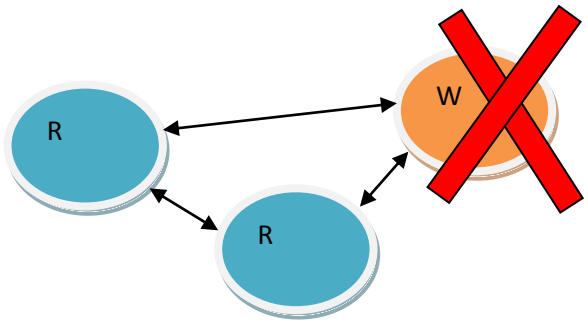
1. ONE - no qual uma operação de leitura retorna os dados da primeira réplica, mesmo o nó sendo o mais antigo. Caso o dado seja antigo, leituras posteriores serão realizadas sobre mais novos. Tal processo é denominado *Read Repair*. O nível de consistência baixa é recomendado quando não há preocupação se o dado é antigo, com a possibilidade de perda, ou o requisito da aplicação é de alto desempenho de leitura. Este caso acontece quando alguns *peers* ficam *offline* antes de ocorrer a replicação;
2. QUORUM - as operações de leitura e escrita podem se propagar para a maioria dos *peers* antes que elas sejam consideradas como sucesso e o cliente seja notificado. O QUORUM é configurado pelo número de leitores R que retornam uma operação de sucesso, o número de escritores W que retornam uma operação de sucesso e o número de *peers* N que participam da replicação de dados. Um fator de replicação ótimo é dado

[19] http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html

quando (R + W) > N. Por exemplo, em um sistema com 3 *peers* um fator de replicação ótimo consiste em R = 2 e W = 2. Neste cenário, pelo menos uma réplica tem a função de escritor e leitor, sendo que quando ocorrer falha em um desses *peers*, a disponibilidade não é afetada com dois nós com a mesma função de escrita.

3. ALL – Neste nível de consistência uma escrita deve ser realizada no *commit log e* em *memtable* de todas as réplicas no cluster para aquela partição. Este nível provê a ocorrência de uma consistência forte por garantir escritas mais atuais em todos os nós em uma partição.

Figura 5.2 – Cenário de falha com N = 3, R = 2 e W = 1



Fonte: O Autor

No caso da Figura 5.2, caso um *peer* escritor (W) venha a falhar, não haverá operação de escrita nos outros nós leitores, deixando o sistema momentaneamente inconsistente até um novo *peer* escritor ser inserido na rede, havendo apenas a disponibilidade de leitura dos dados desatualizados nos servidores *Readers*.

5.2.2. Versionamento dos dados

De acordo com (Lakshman et al., 2009), o sistema utiliza *timestamps* no nível de família de colunas para determinar qual o valor mais recente da informação ao realizar uma requisição de leitura. Porém não há nenhum mecanismo de controle de versão presente no Cassandra, assim como estruturas vectorclock presentes no Dynamo. Para acessar um valor mais antigo de um dado, segundo (Lakshman et a.,l 2009), é preciso definir uma operação de gerenciamento de escrita na SSTable antiga, visto que a inserção dos dados é realizada através de estruturas em memória *memtable* volátil.

5.2.3. Partição do sistema

De forma semelhante com o Dynamo, segundo (Lakshman et al 2009), o sistema Cassandra utiliza *hash consistente* para particionar os dados através de cluster, porém usando uma função de *hash* que preserva a ordem dos nós distribuídos. A cada nó no sistema é atribuído um valor

aleatório, representando sua posição no espaço em forma de anel composto pela identificação das demais réplicas.

Cada item do dado identificado por uma chave é atribuído a um nó pela função de *hash* que realiza um mapeamento para a posição no espaço circular de chaves que representam os nós do sistema. Dessa forma, cada nó se torna responsável pela região entre ele o seu nó predecessor no espaço circular. A principal vantagem de *hash consistente* é que a chegada ou saída de um nó somente afeta seus nós vizinhos, os demais permanecem não afetados.

Diferentemente do Dynamo, o modo de endereçamento do dado é feito analisando a carga de informação no espaço circular, afirmam Lakshman et al. (2009). O número de requisições para nós com baixa carga de trabalho no anel tem o intuito de aliviar o trabalho para nós com grande carga de trabalho. O sistema escolhe este tipo de procedimento para manter a flexibilidade na implementação e ajudar com a escolha de obter balanceamento de carga mais determinístico.

5.2.4. Detecção de Falha

Falha em datacenters ocorrem devido a interrupção de energia, falha de rede, falha de hardware ou causas naturais. Tais falhas causam a indisponibilidade de acesso a servidores, sendo preciso gerenciamento de carga para atribuir a requisição a outro nó disponível. Para lidar com tal problema, o sistema Cassandra utiliza uma versão modificada do Detector de Falha de Precisão (Accural Failure Detector - AFD) (Lakshman et al., 2009). O funcionamento deste detector consiste em um módulo de detecção de falhas que emite um valor, o qual representa o nível de suspeita de falha para cada nó monitorado. O valor deste limiar é constantemente ajustado de acordo com as condições da rede e carga dos nós monitorados.

5.2.5. Caso de uso – Facebook Inbox Search

A infraestrutura de mensagens do Facebook disponibilizada para milhões de usuários, suporta bilhões de mensagens por mês (Facebook 2010). Assim, com este cenário, dois padrões de dados emergiram:

1. Dados temporais que tendem a ser voláteis
2. Um conjunto de dados em crescimento que raramente é acessado

Embora a aplicação utilize a abordagem NoSQL primordialmente pela capacidade de dispor de maior escalabilidade, sistemas de banco de dados relacionais foram utilizados para prover maior segurança e atomicidade nas transações como é o caso do MySQL (Facebook 2010).

Com o aumento crescente no número dessas mensagens, a modelagem do sistema Cassandra foi substituída pelo HBase pela dificuldade em reconciliar novas estruturas de mensagens e o modo de lidar com a consistência eventual. Embora o Cassandra oferecesse suporte de alta escalabilidade e disponibilidade, certas características como balanceamento de carga automático, suporte a compressão, *shards* múltiplos por servidor não eram oferecidas pelo sistema. Por outro lado, além de tais características, o sistema HBase providenciou outras propriedades como

replicação dos dados e o gerenciamento de dados com a técnica de *map-reduce* presente no Haddop[20].

5.3. Teste de consistência e disponibilidade com o Cassandra

Para fins de medir a consistência e disponibilidade do Cassandra, um teste foi realizado com a criação de um cluster com múltiplos nós servidores, realizando operações do lado cliente na rede. A topologia criada consiste das seguintes características descritas no Quadro 5.1.

Quadro 5.1 – Resumo das configurações de teste do sistema

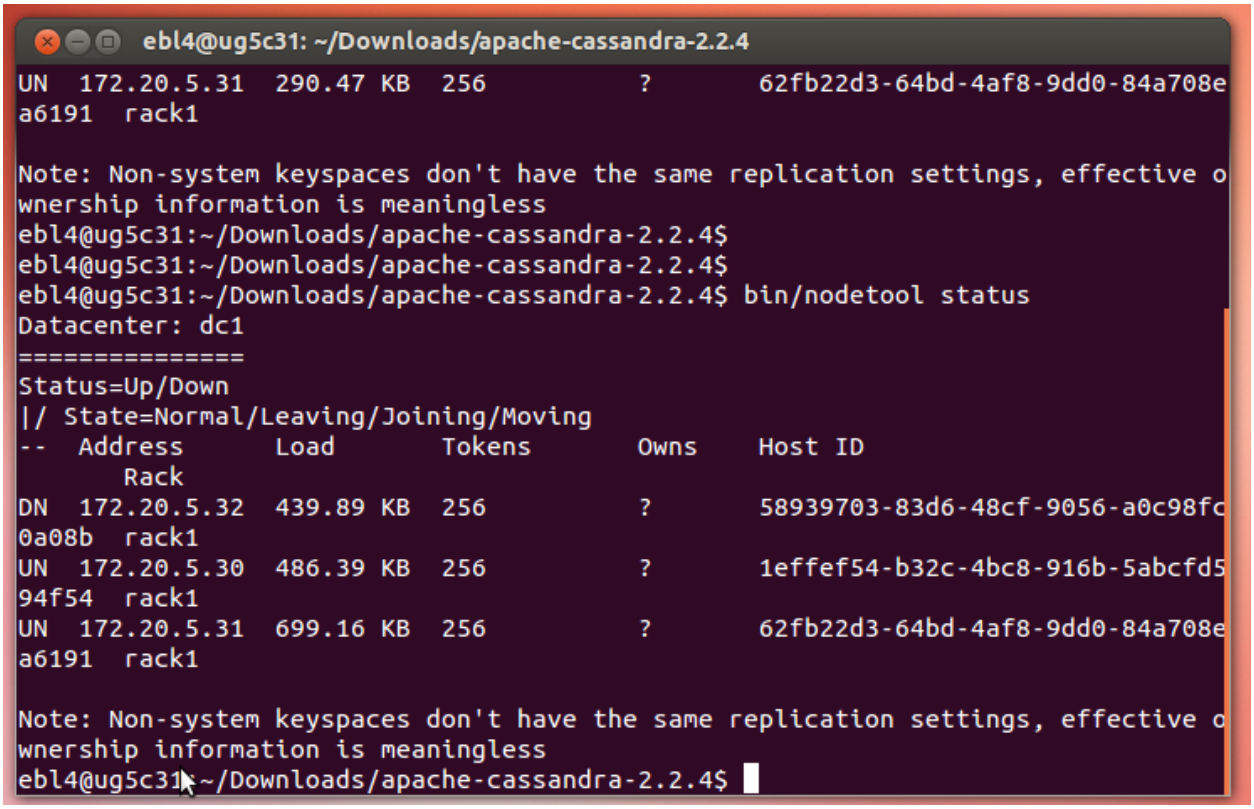
Propriedade	Valor de Configuração	Descrição
Fator de replicação	3	Ver subseção 5.2.1
ConsistencyLevel	ONE	Ver subseção 5.2.1
Estratégia replicação	SimpleStrategy	Estratégia de replicação simples considerando apenas um Datacenter (DC1)
Snitch	GossipingPropertyFileSnitch	Responsável por atualizar automaticamente todos os nós usando o protocolo Gossip para *bootstrapping* ao inserir um novo nó no cluster. Tal configuração do é recomendada para produção.
Réplicas	2	Um nó *seed*: com a responsabilidade de realizar o *bootstrap* de novos nós que são inicializados no cluster. Um nó escravo: responsável por realizar as requisições de operações no banco de dados

Fonte: O autor

[20] https://hadoop.apache.org/

Contudo, de acordo com a documentação atual do sistema pela plataforma de desenvolvimento DataStax[21], é recomendável que a topologia contenha mais de um nó *seed* para permitir tolerância de falhas neste nó. A Figura 5.3 mostra a configuração do cluster com três (3) nós peers, dos quais dois são nós *seeds* e um nó *peer* normal. Cada nó que está executando no cluster deve ter configuração UN (Status – UP, State – Normal). Na Fig. 5.3 apenas o dos nós *seed* não opera no cluster.

Figura 5.3 – Status de cada nó no cluster



Fonte: O Autor

Dois testes foram utilizados quanto à ordem de operações:

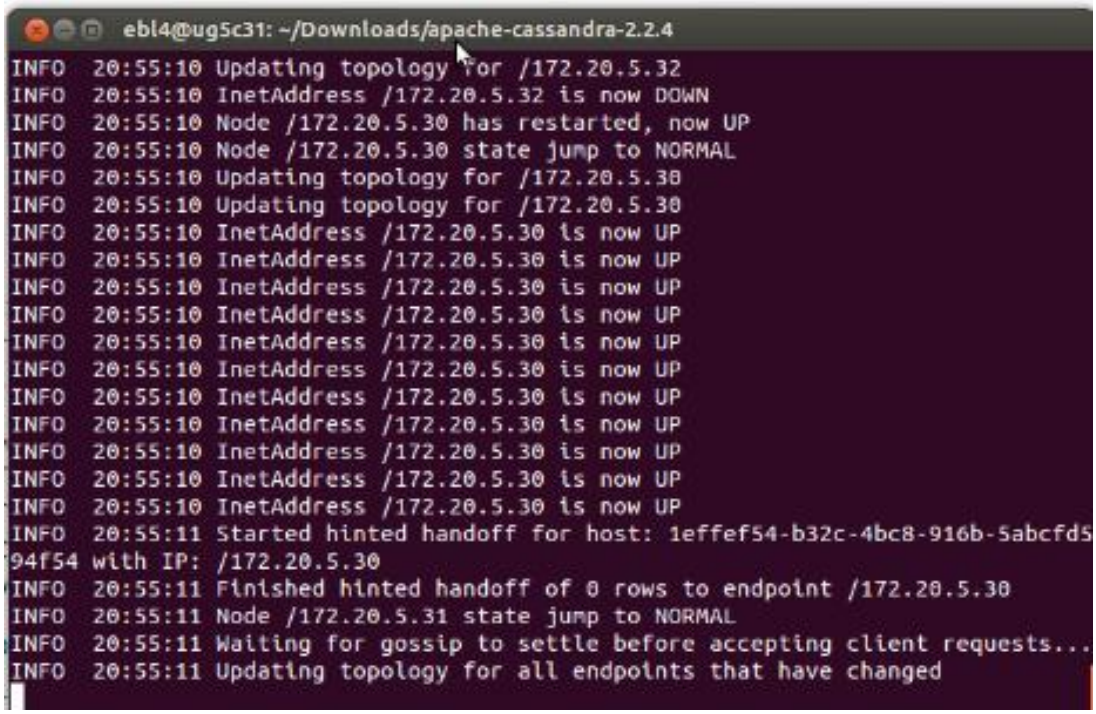
1. **Inserção de dados no nó *seed* -** Com ambos os nós contendo 4 registros de dados em uma tabela, o teste consistiu em realizar um conjunto de 1000 inserções de dados na mesma tabela do lado do nó *seed*; e

[21] http://docs.datastax.com/en/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html

2. **Inserção de dados na réplica** - Com os nós contendo 4 (quatro) registros em uma tabela, foram realizadas 1000 inserções de dados na mesma tabela do nó escravo.

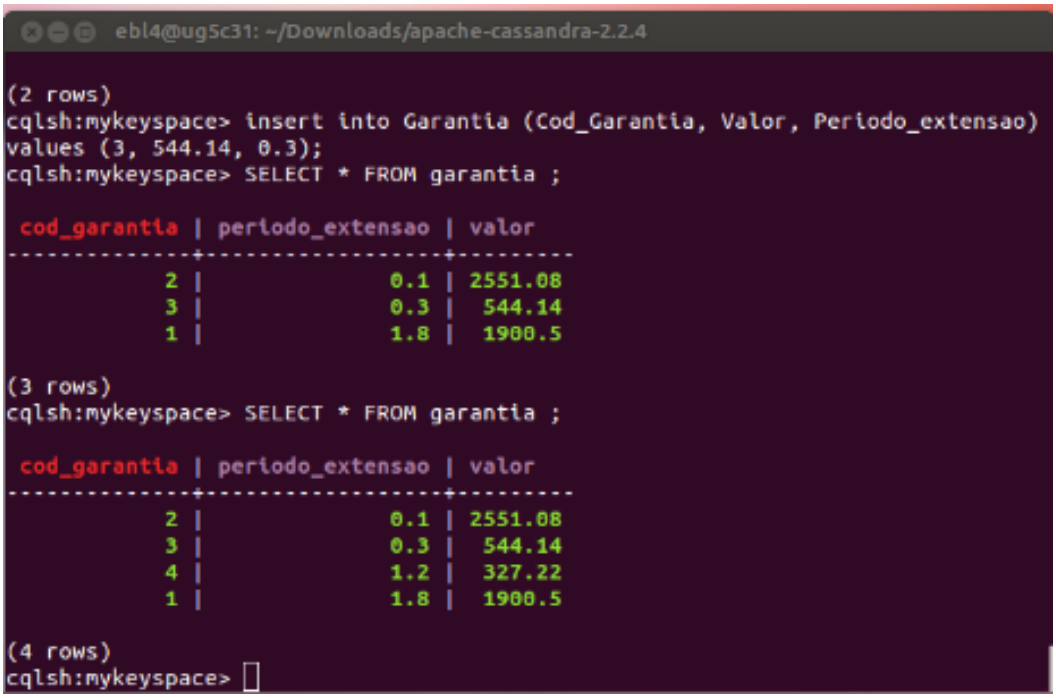
As figuras 5.4 e 5.5 mostram a primeira parte do teste, no qual o *peer* comum foi interrompido, voltando em seguida a ser executado. Deste modo, o banco adquiriu a mesma consistência dos dados em ambos os nós com as inserções no *seed*. Já no segundo teste, o sistema do lado *peer* não persistiu as inserções de dados. Ao finalizar ambos os sistemas nas máquinas e executando novamente, ambas as réplicas voltaram à condição de consistência inicial presente no nó *seed* com quatro registros da tabela (Fig. 5.6).

Figura 5.4 – Reiniciando nó *seed 1* após as transações no *peer*



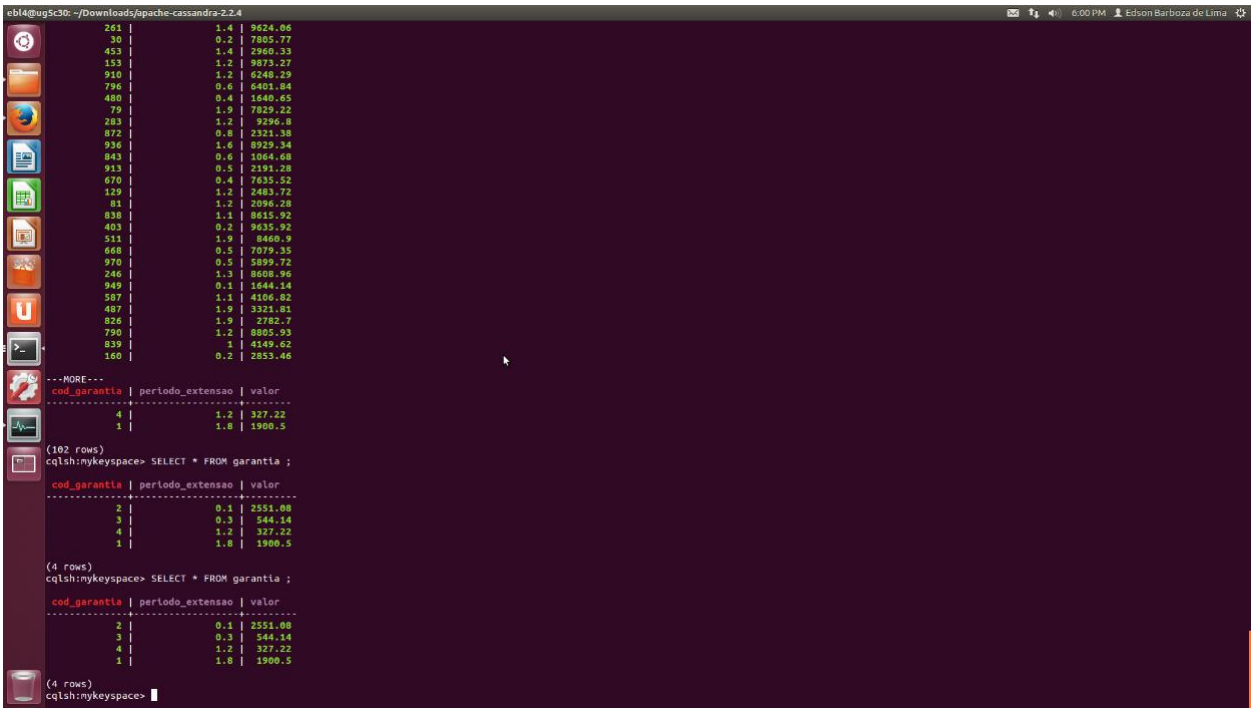
Fonte: O Autor

Figura 5.5 – Consulta no nó *seed* após restabelecer a conexão do cluster



Fonte: O Autor

Figura 5.6 – Consulta no nó *peer* após sua reinicialização.



Fonte: O Autor

5.4. Análise entre sistemas

A análise do mecanismo de funcionamento dos sistemas BigTable, Dynamo e Cassandra expõe os principais conceitos e técnicas de SGBDN, que são utilizados a depender do objetivo da aplicação. As políticas e técnicas apresentadas em cada um dos sistemas descritos nas seções anteriores facilitam o entendimento de outros sistemas de banco de dados em nuvem. Como exemplo, tem-se o sistema Neo4j[22] que utiliza o particionamento dos dados em réplicas *master-slave*. Tal técnica permite requisição de escrita do nó *slave* para o nó *master* e consistência eventual dos dados de acordo com a propagação do *master* para os demais *slaves.*

O sistema BigTable provê uma forte garantia de consistência, chegando a diferenciar dos outros sistemas. Outro fato é que as características CAP não são permitidas nem ajustáveis para os clientes, com efeito negativo sobre a disponibilidade do sistema que recorre a outros mecanismos para complementar essa característica. O Cassandra permite que o usuário defina o nível de consistência a ser utilizado além de conter mecanismo para o usuário tomar a ação de resolver os conflitos que surgem no sistema. Já o sistema Dynamo exige que os conflitos sejam resolvidos pelo próprio cliente.

Em relação ao número de escritas, visando fornecer aspectos de disponibilidade, durabilidade e consistência, o Dynamo opta por balancear o número de réplicas de escrita e de leitura em relação ao número de nós N em cada instância do banco. Por exemplo, com N configurado para 3, a configuração comum (N, R, W) usada por diversas instâncias Dynamo é (3,2,2). As requisições do Cassandra através da técnica de *hash consistente* operando com uma previsão de falha baseada em aproximação por distribuição exponencial, de acordo com o detector de falha AFD. Já o sistema BigTable lida com sistema Chubby para prover disponibilidade do sistema. Neste caso o Chubby utiliza cinco réplicas uma delas sendo a *master* que ativamente serve às requisições. Entre outras atividades, o Chubby também é responsável por assegurar que há pelo menos uma réplica *master* ativa para responder requisições e então encaminhar esta aos servidores *tablet.* Outra atividade consiste em armazenar a localização dos dados em *bootstrap.* Dessa forma, o sistema se torna indisponível caso o Chubby fique indisponível.

A subseção seguinte faz uma comparação entre as técnicas utilizadas pelos sistemas.

5.5. Comparação das técnicas estudadas

Em relação às instâncias do sistema, a análise de aspectos do BDN deve ser realizada ao conjunto desses nós nas partições da rede. Cada técnica pode interferir de maneira positiva ou negativa nas características CAP. Por isso, sistemas com características semelhantes devem ser analisados sob o mesmo enfoque de escalabilidade em relação às propriedades CAP, diferentemente de sistemas que complementam a aplicação com abordagem tradicional como exemplificado por SGBDR na Seção 5.2.5.

[22] http://www.neo4j.com

A técnica primordial quanto ao acesso em escrita e leitura dos dados foi a replicação de cada sistema utilizando a abordagem *master-slave*, na qual uma réplica é eleita para coordenar os outros nós que devem resolver a requisição a ser tratada. Abordagens de recuperação de falhas indicam a eleição de um novo nó *master* caso este fique indisponível. Assim, um dos nós *slaves* deve assumir a função do *master* de acordo com certas prioridades definidas em cada réplica, dependendo de características como capacidade de processamento e memória. O BigTable, no entanto, utiliza o sistema Chubby para lidar com a replicação do sistema. O sistema, dessa forma, depende da disponibilidade do Chubby para manter as réplicas funcionando.

Outra técnica importante quanto a CAP é a abordagem do *hash consistente* para particionar os dados. No Cassandra, quando um nó inicia pela primeira vez, um valor aleatório de token é gerado para a posição do nó no espaço circular de chaves. Quando a requisição de leitura/escrita chega a qualquer nó no cluster uma máquina de estado identifica o nó responsável pela chave do valor a ser alterado, encaminha a requisição ao esse nó e espera até a resposta de confirmação que a requisição chegou. O sistema assume que houve falha caso a requisição não chegue ao nó que inicialmente encaminhou a mensagem. Por fim, a requisição retorna ao cliente disponibilizando a última atualização do dado de acordo com o *timestamp*. O sistema programa um reparo dos dados em qualquer réplica caso elas não contenham o último valor do dado.

Já no Dynamo, é utilizada uma versão modificada do *hash* consistente. Ao invés de mapear um nó a um ponto simples no sistema, cada nó é atribuído a múltiplos pontos no anel circular da DHT. Para este propósito, o Dynamo utiliza o conceito de nós virtuais. Cada nó virtual é semelhante a um nó simples, como descreve a seção 4.1.2, porém cada nó pode ser responsável por mais de um nó virtual. Múltiplas posições no espaço circular de chaves são atribuídas a um nó quando um ele é adicionado ao sistema. Caso um nó fique indisponível, a carga de trabalho deste nó é dispersada para os nós restantes disponíveis. Quando um nó volta a ficar disponível novamente, ou um novo nó é adicionado ao sistema, o nó mais novo aceita a mesma carga de requisições atribuídas aos nós restantes.

6. CONCLUSÃO

Embora algumas aplicações utilizem sistemas de banco de dados para prover certas características inerentes ao ambiente em nuvem, o trabalho mostra a dificuldade de escolher um sistema que apresente uma solução mais próxima à ideal que obedeça às restrições CAP. Toda técnica que afeta uma característica de forma positiva, tem efeito negativo sobre os outros dois, ou na latência do BDN. Modificações em uma técnica podem trazer ganhos ou perdas de propriedades ao sistema.

Empresas atuais, no entanto, investem em sistemas de múltiplos banco de dados com abordagem mesclada com SGBDR e banco de dados NoSQL de acordo com a necessidade de cada aplicação. Aspectos de consistência e disponibilidade devem ser tratados em termos de escalabilidade. Outra característica importante, além da resolução de conflitos dos dados, é a definição por parte do usuário das responsabilidades CAP com o sistema, quando níveis de consistência podem ser ajustados pelo próprio cliente.

Os testes com o sistema Cassandra mostraram que é preciso definir balanceamento das requisições pelo nó *master*, sendo a réplica que gerencia as requisições no sistema. Quanto à latência do sistema, o sistema mostrou-se regular em termos de atualização dos dados no nível de consistência básico ONE e fator de replicação alto em relação ao número de réplicas. As estratégias de replicação compõem características interessantes quanto à disposição dos datacenters no cluster com múltiplos nós, abordando estratégia simples de replicação em apenas um datacenter ou estratégias com maior escalabilidade e particionamento em rede com mais de um datacenter.

6.1. Limitações do trabalho

Considerando aspectos de modelo de dados, o presente trabalho abordou o teste em banco de dados NoSQL baseado em coluna. Os testes com o ambiente citado na Seção 5.3 foram limitados a poucos recursos computacionais e disponibilidade de aplicação em ambiente restrito à carga de dados. As configurações dos testes não abordam políticas de sistema em larga escala de produção, e restrições de medidas de grandes quantidades de dados.

6.2. Trabalhos futuros

Trabalhos futuros podem contribuir com testes de disponibilidade em outras categorias de SGBDN como análise de desempenho e latência do sistema utilizando maior recurso computacional. Por fim, é de grande importância analisar o impacto que estruturas com índices secundários causam em relação ao desempenho de certos SGBDN.

REFERÊNCIAS BIBLIOGRÁFICAS

10 of the most useful cloud databases – Disponível em http://www.networkworld.com/article/2162274/cloud-computing/10-of-the-most-useful-cloud-databases.html >. Acessado em 13/11/2015

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley.

ASP - http://www.asp.net/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/data-partitioning-strategies. Acessado em 17/11/2015

Avinash Lakshman e Malik Prashant. Cassandra: A Decentralized Structure Storage System. Facebook. 2009

Baldoni, R., & Raynal, M. "Fundamentals of distributed computing: Apractical tour of vector clock systems." 2002

Carmem. Sistemas de gerenciamento de dados na nuvem. http://www.inf.ufpr.br/carmem/presentation/erbd2011-9pp.pdf . Acessado em 17/11/2015

Chandra, T., Griesemer R., Redstone, J. Paxos. Made Live - An Engineering Perspective. 2007

Coulouris, G., Dollimore, J., Kindberg, T., Blair, G. Sistemas Distribuídos: Conceitos e Projeto. 5 ed. Estudo de caso: Google. 2013

DeCandia, G., Hastorun, D., Jampani,M., Kakulapati, G., Laksh-man, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev.

Elmasri, R., Navathe, S.. Sistemas de Banco de Dados. 4 ed. Presley. 2005

Erick Brewer, "Cap Twelve Years Later: How the Rules Have Changed". 2012

Facebook - https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919/. Acessado em 08/12/2015

Fay Chang, Jeffrey Dean, "Bigtable: A Distributed Storege System for Structure Data". 2006

Felipe Gutierrez - <https://sites.google.com/site/lipe82/Home/diaadia/columndatabases > Acessado em 23/11/2015.

Freitas, E.L.S.X. Um Modelo para Garantia de Consistência dos Dados em Sistemas de Banco de Dados Relacionais em Nuvem. Centro de Informática – UFPE. 2015

Haerder, Theo, and Andreas Reuter. "Principles of transaction-oriented database recovery." 1983.

http://200.17.137.109:8081/novobsi/Members/josino/fundamentos-de-banco-de-dados/2012.1/Gerenciamento_Dados_Nuvem.pdf. Acessado em 01/11/2015

http://aws.amazon.com/pt/running_databases/#relational_amis> Acessado em 13/11/2015.

http://docs.basho.com/riak/latest/theory/comparisons/cassandra/. Acessado em 07/12/2015.

https://www.quora.com/What-are-the-disadvantages-of-using-BigTable-over-MySQL-PostgreSQL-or-MongoDB. Acessado em 22/11/2015.

Karger, David, et al. "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web." 1997

Limits in DynamoDB - http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. Google. 2006

Moreira, V. Distributed Hash Table. Universidade de São Paulo. Acessado em 23/11/2015

Pinheiro, D.S.M. Análise Comparativa sobre a consistência de banco de dados na nuvem. Centro de Informática – UFPE, 2013.

Pramod J. Sadalage, Martin Fowler. NoSQL: A brief guide to the Emerging World of Polyglot Persistence. 2013

Ricardo Puttini, Thomas Erl. Cloud Computing: Concepts, Technology & Architecture. 2013

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak ."The Google FileSystem". Google 2003.

Tim Gross. http://0x74696d.com/posts/falling-in-and-out-of-love-with-dynamodb-part-ii/. Acessado em 08/12/2015

Jenny Zhang. 2014. NoSQL – HBase vs Cassandra vs MongoDB - http://jennyxiaozhang.com/nosql-hbase-vs-cassandra-vs-mongodb/. Acessado em 13/01/2016