Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação 2012.2

ANÁLISE COMPARATIVA SOBRE A CONSISTÊNCIA DE BANCO DE DADOS NAS NUVENS

Trabalho de Graduação

Dário Saraiva de Melo Pinheiro

Recife, 30 Abril de 2013.

Universidade Federal de Pernambuco

Centro de Informática

Dário Saraiva de Melo Pinheiro

ANANÁLISE COMPARATIVA SOBRE A CONSISTÊNCIA 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

Recife, 30 Abril de 2013.

Dedico este trabalho a Dário Filho e Telma Pinheiro, meus amados pais.

RESUMO

Sistemas nas nuvens têm se tornado cada vez mais populares, com vantagens claras como, por exemplo, ser extremamente escalável. Todavia, sistemas nas nuvens também possuem suas limitações.

Segundo o teorema CAP [14], um sistema computacional distribuído é capaz de garantir simultaneamente no máximo duas das três características a seguir:

1. Consistência (C de *Consistency*);

2. Disponibilidade (A de *Availability*); e

3. Tolerância à partição da rede (sistema continuar funcionando mesmo após falhas de uma das partes), (P de *Partition tolerance*).

Tendo essa restrição, as soluções de Banco de Dados (BD) nas nuvens atuais, em sua maioria, optam por relaxar os aspectos de consistência em favor da Disponibilidade e Tolerância à partição da rede. Cada sistema de BD nas nuvens garante um nível de consistência diferente e, para permitir tal característica, cada um possui políticas e mecanismos distintos. Este trabalho tem como objetivo analisar as técnicas empregadas pelos principais Sistemas de Bancos de Dados nas Nuvens atuais, que afetam diretamente as características CAP.

Palavras-chave: *SGBD*, *Computação nas Nuvens*, *CAP, NoSQL*.

ABSTRACT

Cloud systems are becoming increasingly popular because they can provide great advantages such as being very scalable. However, cloud systems also have their limitations.

According to the CAP theorem [14], a distributed computing system is able to ensure simultaneously at most two of the following three characteristics:

1. Consistency;

2. Availability; and

3. Partition Tolerance.

With this restriction, Cloud Database Solution mostly opts to relax aspects of consistency in favour of Availability and Partition Tolerance. Each Cloud DB system ensures a different level of consistency and to allow such different characteristic each one has different policies and mechanisms. This paper aims to analyze the techniques currently used by major cloud databases that directly affect the CAP characteristics.

Keywords*: DBMS, Cloud Computing, CAP, NoSQL.*

LISTA DE ILUSTRAÇÕES

SUMÁRIO

CAPÍTULO 1: INTRODUÇÃO

A popularização da Internet e a explosão de dados, em conjunto com a criação de vários centros de processamento de dados (*datacenters*) por empresas diversas, fizeram surgir um novo conceito: Computação nas Nuvens. Uma subdivisão desse conceito é Banco de Dados nas Nuvens (BDN).

BDN apresenta vantagens e desafios diferentes de sistemas de bancos de dados convencionais. Para de fato usufruir das vantagens do conceito da nuvem, um BDN precisa ser distribuído, o que acarreta o problema dele ser incapaz de cumprir ao mesmo tempo três características desejáveis de um Sistema de Gerenciamento de Banco de dados (SGBD): Consistência, disponibilidade e tolerância à partição de rede [12, 14]. Essa questão será contextualizada com mais detalhes nas seções 1.1 e 1.2.

1.1 Contexto da nuvem

Para definir o que é um Banco de Dados nas Nuvens, primeiro é preciso definir o que é computação nas nuvens. Na literatura existem varias definições do termo, a que será usada foi definida pelo Instituto Nacional de Padrões e Tecnologia dos EUA:

> "Sistema Computacional nas Nuvens é [...] um modelo que permita, via rede, acesso a recursos computacionais compartilhados e configuráveis (exemplo: servidores, recursos de armazenamento, aplicações e serviços) que podem ser provisionados e liberados requerendo mínimo esforço administrativo, e mínima interação com o provedor do serviço" [22].

Em outras palavras, computação nas nuvens é um modelo que permite que clientes utilizem recursos físicos ou virtuais como um serviço, através da rede. Dessa forma, uma das poucas requisições para um sistema ser considerado um BDN é que ele ofereça recursos que facilitem a persistência e recuperação de dados, através da rede, e que ele exponha essas funções na forma de serviços. Mas esses seriam apenas os requisitos mínimos para um BDN, considerando que eles

são operados dentro de um centro de processamento de dados (*datacenter*) e podem ter acesso a inúmeras máquinas, um BDN eficiente é projetado para utilizar esses recursos físicos.

1.1.1 Banco de dados Convencionais

Sistemas de Gerenciamento de Banco de Dados (SGBD) convencionais atuam em apenas uma máquina ou num conjunto pequeno de máquinas com localização física próxima uma das outras. Nesse contexto, o modelo de SGBD mais popular é o Relacional (SGBDR) [15]. Ele permite a execução de transações com propriedades de Atomicidade, Consistência, Isolamento e Durabilidade (ACID), além de permitir restrições complexas sobre dados. Embora esse modelo possa ser usado como um BDN, como definido na seção anterior, ele não administra bem um número grande de máquinas, logo ele não aproveita adequadamente um dos maiores potenciais de Computação nas Nuvens: a escalabilidade.

1.1.2 A restrição CAP

Um BDN eficiente precisa utilizar-se de mais de uma máquina física para ser eficiente. O teorema CAP passa a ser relevante no desenvolvimento de um BDN por este motivo. O CAP afirma que um sistema computacional distribuído é capaz de garantir simultaneamente no máximo duas das três características a seguir [12, 14]:

1. Consistência (C de Consistency);

2. Disponibilidade (A de Availability); e

3. Tolerância à partição da rede (sistema continuar funcionando mesmo após falhas de uma das partes), (P de Partition tolerance).

Um cenário simplificado que exemplifica esse teorema está descrito abaixo.

Um sistema distribuído qualquer possui dois nós ‘A’ e ‘B’ que estão incomunicáveis por questão de partição na rede. Eles podem receber requisições para mudar de estado. Para o sistema permanecer consistente, um nó precisa receber uma informação quando o outro for alterado. Dessa forma, quando o nó ‘B’

receber uma requisição para alterar seu estado, os seguintes cenários são possíveis:

1. ‘B’ altera seu estado atual e ignora ‘A’ (ou envia uma mensagem assíncrona que nunca chega a ‘A’), tornando o estado do sistema inconsistente;
2. ‘B’, para preservar a consistência ignora a requisição recebida, tornando o sistema indisponível para aquela mensagem ou cliente; ou
3. O sistema só conseguiria se manter disponível e consistente se não houvesse partição, ou seja não é tolerante a partições.

1.2 Objetivo do trabalho

Considerando o que foi descrito na seção 1.1 e que quando os BDN são distribuídos em inúmeros nós distintos, a probabilidade de ocorrer partição em algum ponto da rede é alta, todo BDN realmente distribuído precisa fazer uma escolha entre relaxar a consistência ou a tolerância à partição da rede. A escolha de qual dos aspectos relaxar e o quanto relaxar, são questões não triviais e podem variar dependendo do tipo de problema que o sistema esta tentando solucionar.

Este trabalho tem como objetivo analisar os BDN atuais e as técnicas que os mesmos utilizam para mitigar o problema descrito. A fim de expor as vantagens e desvantagens de cada técnica, possibilitando a identificação de conjuntos das mesmas que melhor se adequa a diversos cenários.

1.3 Organização do documento

O Capítulo 2 apresenta algumas considerações sobre sistemas distribuídos, além de técnicas gerais empregadas por Sistemas de Banco de Dados (SGBD) nas Nuvens e a razão da escolha dos sistemas que serão explorados em detalhes nos capítulos seguintes. Os capítulos 3 a 5 detalham os sistemas PNUTS, Dynamo e BigTable. O capitulo 5 apresenta as conclusões do presente trabalho e sugestões de trabalhos futuros.

CAPÍTULO 2: BANCO DE DADOS NAS NUVENS

A fim de entender melhor as políticas de um BDN especifico e suas peculiaridades, é necessário conhecer o contexto geral de SGBDN e as possíveis técnicas que podem ser empregadas pelos mesmos. Este capítulo apresenta considerações básicas sobre SGBDN que costumam divergir de sistemas convencionais; em seguida explora os principais tipos de técnicas usadas pelos BDN para minimizar as consequências negativas impostas pelo teorema CAP; e por fim apresenta os sistemas que serão explorados com mais detalhes neste trabalho.

2.1 Linguagens de consulta

Embora existam BDN que utilizem SQL, a maior parte das linguagens utilizadas encontra-se na categoria NoSQL [30]. Como não existe um padrão NoSQL, as linguagens de consulta podem variar entre si. Isso permite que os SGBDN tenham certa liberdade na definição de seus comandos, de tal forma que alguns sistemas possuem comandos que expõem ao cliente níveis distintos de CAP.

2.1.2 Latência

Latência é o tempo que leva para um componente receber uma resposta de uma requisição feita [20]. A latência do sistema é um aspecto com forte ligação à característica de tolerância à partição da rede. Um sistema distribuído qualquer precisa saber distinguir entre um sistema falho ou inalcançável e um sistema que está apenas demorando a responder.

2.1.3 Diferenças em nomenclaturas e definições

SGBDR clássicos garantem que uma transação possui características ACID [15] independente do numero de transações sendo feita paralelamente:

1. Atomicidade - o sistema executa a transação com sucesso ou simplesmente não a executa (sem efeitos colaterais de execuções parciais ou que gerem erro);
2. Consistência - as regras de integridade do banco são respeitadas;

3. Isolamento - os efeitos de uma transação não finalizada não são expostos às demais; e
4. Durabilidade - caso a transação seja efetuada com sucesso, as alterações no banco devem persistir.

O conceito de consistência do acrônimo ACID difere dos utilizados nos SGBD distribuídos das seguintes formas [12, 14, 15, 27]:

1. Unidade de Consistência - A noção clássica de consistência se refere ao banco de dados todo, considerando todos os objetos do banco. A comunidade que trabalha com bancos distribuídos normalmente considera consistência de um objeto único no banco;
2. Consistência, do ponto de vista do sistema e do cliente - A noção clássica considera a consistência dos dados vista do SGBD, isto é, os dados estão consistentes se os mesmos estão persistidos no banco de forma consistente, ou seja, os efeitos de operações paralelas no banco devem ser iguais aos efeitos se as mesmas fossem feitas em sequencia. Em contraste, no meio de bancos distribuídos a consistência é considerada do ponto de vista do cliente, ou seja, os dados estão consistentes se os dados que um cliente tem acesso são consistentes, independente do seu estado real de persistência no banco.

2.2 Aspectos e Técnicas gerais de SGBD nas Nuvens

Um BDN pode implantar diversas politicas e técnicas que têm impacto grande nas suas características CAP, esta seção abordará os aspectos dos BDN que costumam se de diferenciar mais, e tem um forte impacto em outras características do sistema.

2.2.1 Modelo de dados e sistema de persistência

Um contraste grande que os SGBD nas nuvens têm em relação aos sistemas comuns é o modelo de dados e o sistema de persistência.

Enquanto que o modelo de dados em sistemas convencionais é, em sua maioria, o modelo relacional, em sistemas nas nuvens ocorre um sistema de chave-valor ou então de tabela multidimensional. Existe também o conceito de modelo orientado a documento, mas muitas vezes o acesso aos dados nesse modelo é considerado equivalente ao de chave-valor, como, por exemplo o CouchDB [19].

O nível de responsabilidade e complexidade do sistema de persistência em relação à CAP varia entre os SGBD. Vai desde quase nenhuma garantia, até garantias complementares ao SGBD distribuído em questão.

A Figura 1 mostra a classificação de alguns SGBD pelo seu modelo de dados e sistema de persistência.

Figura 1- Classificação de SGBD por sistema de persistência

Sistema	Modelo Dados	Sistema Persistência
PNUTS	Chave-Valor	RSGBD convencionais Variáveis
DYNAMO	Chave-Valor	RSGBD convencionais Variáveis
BIGTABLE	Tabela Multidimensional	Google File System (GFS)
HBASE	Tabela Multidimensional	Hadoop Distributed File System (HDFS)
CASSANDRA	Tabela Multidimensional	Memoria Secundaria

Fonte: Adaptado de Santhosh Kumar Gajendran [29].

2.2.2 Particionamento dos Dados

Particionar os dados costuma trazer benefícios grandes para a escalabilidade do sistema. Todos os sistemas analisados utilizam partição em algum momento de sua execução ou dependem de algum sistema externo que faz uso intensivo de tal técnica.

Quando o próprio Sistema de Banco de dados nas Nuvens (SGBDN) utiliza-se de partição, ele atribui a cada nó uma lista de valores, e o nó em questão passa a ser responsável por todo e qualquer dado que possua o valor de chave nessa lista. Considerando que a lista pode ser selecionada arbitrariamente, a parte gerenciadora

do sistema pode escolher um conjunto de valores de forma inteligente (levando em conta, por exemplo, as características físicas do nó). Duas formas comuns de fazer isso são [6, 8, 23]:

1. Partição por intervalo - O processo consiste simplesmente em dividir o espaço das chaves em intervalos, e atribuir a cada nó um intervalo distinto; e.
2. Partição por *Hash* - Similar à partição por intervalo, porem uma função *Hash* é utilizada na chave.

A técnica pode ser aplicada mais de uma vez, por exemplo, o sistema Hive [31] particiona suas tabelas, e cada partição pode ser particionada mais uma vez pelo sistema de persistência HDFS.

A utilização de partição pode auxiliar o sistema a manter cada uma de suas partes com uma carga de trabalho similar. O Cassandra [2], por exemplo, monitora as requisições feitas para os servidores, e modifica o espaço de particionamento caso seja necessário. Já o Dynamo [10] distribui o espaço de chaves de forma uniforme entre os servidores e assume que as requisições serão em média proporcionais para cada partição.

Cada sistema utiliza uma nomenclatura própria para nomear cada partição, por questões de uniformidade, este texto se referirá às partições de maior granularidade dentro de um sistema, como *tablet*.

2.2.3 Replicação

A replicação nesse contexto engloba tanto a politica usada para replicar um dado, a quantidade de réplicas utilizadas, as características das réplicas (se existe diferenciação de responsabilidades entre as mesmas) bem como a técnica usada para manter as réplicas consistentes entre si.

A forma com que o sistema cria e trata as réplicas é um dos aspectos de maior impacto quanto à CAP [12, 27] A decisão mais simples que pode ser feita é selecionar a quantidade de cópias para cada dado e se ela será feita de forma síncrona ou assíncrona. Essa escolha tem efeito direto em praticamente todas as

características do banco. Um número maior de réplicas pode significar maior disponibilidade e menor latência em sacrifício de consistência, se for implementada uma comunicação assíncrona entre as réplicas, mas pode significar exatamente o contrário (nas operações de escrita) em um sistema de replicação síncrona.

Outro aspecto importante sobre a técnica de replicação é em que nível da topologia física ela atua [2]:

1. Em cada nó físico de menor granularidade;
2. No nível de *datacenter*; ou
3. Ignora a topologia física.

O momento e a forma de resolver conflitos entre as réplicas é outro fator importante com relação à replicação [2, 12]. Por exemplo, o sistema Dynamo resolve conflitos quando executam operações de leitura, já Cassandra possui um método de escrita segura que simplesmente falha caso a escrita fosse gerar divergência entre as réplicas.

2.2.4 Versionamento

A existência de versões diferentes pode ser relevante ou não quanto à CAP, de uma forma ou de outra, o termo “versionamento” será empregado de maneira abrangente neste trabalho com relação às politicas utilizadas por cada SGBD na utilização de versões distintas dos dados. Dois exemplos de versionamento [2, 10, 13]:

1. Versionamento desejável e com pouco impacto à CAP - Sistemas de tabelas multidimensionais mantêm versões antigas dos dados propositalmente, a fim de serem utilizadas pelos clientes. Quando o cliente acessa o dado, ele sempre acessa a versão mais recente do mesmo a não ser que explicite que queira receber uma versão mais antiga; e
2. Versionamento indesejável e relevante à CAP - Sistemas com fraca garantia de consistência podem estar num estado inconsistente num certo tempo, e ter versões distintas do dado espalhados por diversos nós,

precisando aplicar outras politicas complexas para decidir qual versão usar, ou simplesmente entregar qualquer uma ao cliente.

2.2.5 Detecção e recuperação de falhas (partição na rede)

Um aspecto importante em qualquer sistema distribuído é a identificação e recuperação de falhas. Cada sistema possui suas peculiaridades, mas em geral eles costumam seguir uma das duas abordagens abaixo [29]:

1. Coordenado - existe uma entidade que coordena os nós do sistema e é responsável por detectar as falhas e recuperá-las. Obviamente, o próprio coordenador pode falhar. Por isso ele normalmente possui uma máquina de *backup,* ou o sistema se encarrega de apontar um novo coordenador dentre os nós restantes automaticamente; e
2. Não coordenado - cada nó é autônomo e consegue identificar se os nós relevantes para ele estão disponíveis ou não.

2.2.6 Características não abordadas

Existem características que tornam sistema de banco de dados nas nuvens mais atraentes, mas não serão abordadas profundamente, pois estão fora do escopo deste texto.

Um exemplo é a técnica MapReduce [9], uma das responsáveis pelo sucesso do sistema BigTable [13] e do HadoopDB [1], que permite um acesso e processamento de quantidades enormes de dados de forma eficiente. Embora seja uma técnica muito utilizada em conjunto com o BigTable, ela não o compõe e também não tem efeito sobre as características exploradas neste texto.

2.2.7 Foco de cada sistema

Considerado o CAP, cada SGBD distribuído pode ser classificado pelas suas características predominantes. A Figura 2 mostra uma classificação de alguns sistemas quanto as suas características predominantes; na Figura 2, o ‘x’ representa que o sistema está classificado nessa categoria, e o ‘-’ que o sistema não possui tal classificação.

Figura 2 - Classificação de SGBD distribuídos

Sistema	ACID ou CA	BASE ou AP	CP	Variável em tempo de execução
Bigtable	-	-	x	-
HBASE	-	-	x	-
HIVE	-	-	x	-
Amazon RDS	x	-	-	-
SGBD relacional convencional	x	-	-	-
CouchDB	-	x	-	-
SimpleDB	-	x	-	-
Amazon S3	-	x	-	-
Dynamo	-	x	-	-
PNUTS	-	-	-	x
CASSANDRA	-	-	-	x

Fonte: O Autor

Sistemas que empregam ACID são mais fortes em consistência e disponibilidade.

Sistema classificados como BASE (Basically Available, Soft state, Eventual consistency) [26] têm menos ênfase à consistência e mais aos aspectos AP. Esses sistemas garantem consistência eventual, ou seja, o sistema pode ter momentos de inconsistência para cada objeto, mas eventualmente eles se tornaram consistentes.

O sistema Dynamo por padrão tem forte disponibilidade e tolerância à partição de rede, mas é possível ao se criar uma nova base Dynamo mudar esse comportamento. Ele é classificado como um sistema BASE [10] em sua descrição, por isso essa classificação será usada neste texto.

2.3 PNUTS, BigTable e Dynamo

Nos capítulos seguintes serão exploradas as características especificas e mais relevantes de alguns sistemas. Será dada uma ênfase maior nos sistemas PNUTS, Dynamo e BigTable, já que estes possuem implementações e conceitos bem distintos entre si, além de abrirem mão de aspectos diferentes do CAP.

Dynamo e BigTable serão descritos mais adiante, pois seus conceitos influenciaram a implementação de inúmeros outros sistema de banco de dados distribuídos, como por exemplo HBase e Cassandra.

PNUTS é um sistema que utiliza conceitos simples e proporciona uma interface onde o cliente pode decidir, em tempo de execução, níveis de consistência variáveis.

Uma característica comum entre os três sistemas é que eles são soluções proprietárias, dessa forma o acesso a suas especificações é limitado. Por isso, embora todas as informações apresentadas nos seguintes capítulos tenham sido extraídas de publicações dos próprios desenvolvedores dos sistemas, existe a possibilidade de que alguma metodologia específica esteja desatualizada em relação a sua versão comercial atual.

2.4 Outras considerações

Este capítulo discute alguns conceitos comuns entre os BDN, além de deixar claro que existe uma gama de possibilidades onde um dado BDN pode variar seu comportamento para atingir diferentes níveis de garantias CAP.

O capítulo seguinte irá explorar as políticas do sistema PNUTS sobre essa ótica.

CAPÍTULO 3. PNUTS

Este capítulo explora o sistema PNUTS [3, 5]. Primeiramente é descrito o objetivo por trás da criação do sistema e sua estrutura. Na sequência é discutido suas principais técnicas que influenciam as características CAP. Por fim o capítulo mostra outras técnicas adotadas pelo sistema que funcionam em paralelo com as técnicas principais.

3.1 Visão Geral do sistema

Platform for Nimble Universal Table Storage (PNUTS) é um sistema criado pela Yahoo! [5]. Foi desenvolvido para suprir as necessidades que a empresa possuía de um sistema de administração de dados distribuído geograficamente, que fosse escalável, de alta disponibilidade e baixa latência [3, 16]. Pela sua natureza, o maior volume das operações no sistema é de leitura de dados. O sistema incialmente foi otimizado para leituras de arquivos pequenos, como *medatados* utilizados por sistemas de redes sociais, ou informações de status de usuário. Apesar de utilizar técnicas simples, ele é um sistema muito interessante por permitir níveis variáveis de consistência em tempo de execução.

Os dados do PNUTS são distribuídos e replicados no nível de tabela em vários *datacenters* espalhados pelo mundo, sendo cada *datacenter* distinto chamado de região (*region*). Cada região possui varias unidades de armazenamento, roteadores e um sistema chamado controlador de *tablets* (*tabletcontrollers)* que organizam as unidades de armazenamento [25].

Cada réplica de uma tabela está completamente contida dentro de um *datacenter*, logo, uma requisição de leitura pode ser feita a qualquer uma das regiões que possua uma réplica. Replicação, atualização e inserção de dados também podem ser feitas em qualquer região que possua a tabela. A propagação de tais operações é feita de forma assíncrona através de outro sistema chamado Yahoo Message Broker (YMB) [5]. De forma geral, considerando as características acima, as requisições de um dado cliente são normalmente feitas à região que se encontra a menor distância do cliente, a fim de diminuir a latência [5].

O sistema garante que cada entrada individual possui consistência eventual, e que cada atualização é aplicada na mesma ordem em cada réplica.

3.2 Arquitetura do sistema

Para entender e analisar as técnicas utilizadas pelo PNUTS é necessário conhecer as estrutura nas quais as técnicas atuam ou dependem, esta Seção irá expor essas estruturas.

3.2.1 Modelo de dados

Os dados do sistema são organizados em tabelas, cada tabela pode ser particionada, cada partição é chamada de *tablet.* Os tablets que compõem uma tabela podem estar espalhados por vários nós do sistema desde que eles pertençam ao mesmo *datacenter*, e um *tablet* individual não esteja em mais de um nó. Os *tablets* são persistidos utilizando SGBD convencionais como MySQL ou InnoDB [3, 5].

Cada linha numa tabela é chamada de registro (record) , e consiste de uma chave, metadado *(*metadata*)* e um valor. O valor pode ser tipos normais com suporte em SQL, mas para manter a flexibilidade do sistema, normalmente são usadas cadeias de bytes que representam um objeto *json* auto descritivo. A chave é apenas um identificador único para cada registro. As informações relevantes do metadado serão discutidas na seção 3.3. A especificação do objeto *json* é variável dependendo de cada tabela e é definido pelo cliente. Esse modelo permite a criação de esquemas diversos, mas não tradicionais, pelos usuários sem afetar o funcionamento do PNUTS [3, 5].

3.2.2 Tablet Controllers

Cada *tabletcontroller* está contido dentro de uma região, e é responsável pela partição das tabelas, bem como pelo mapeamento de cada *tablet* nas unidades de armazenamento físico. Por ser uma parte crítica do sistema, há uma máquina de *backup* para rodar tal operação no caso de falha [3, 5].

3.2.3 Roteadores

Embora os roteadores sejam ignorados nas especificações dos outros sistemas, esses foram inclusos na especificação do PNUTS, pois possuem um comportamento peculiar em caso de falha na rede. Os roteadores são responsáveis para encaminhar uma requisição de usuário para a unidade de armazenamento correta. Para exercer sua função corretamente, os roteadores copiam periodicamente as informações de mapeamento feitas pelos *tabletscontrollers* e as armazenam em memória volátil. Caso o roteador não consiga resolver uma requisição do cliente por conta de partição da rede ou o seu mapeamento em memória estar desatualizado, uma mensagem de erro é retornada ao cliente, o roteador reinicia automaticamente, e nenhum processo reparativo é feito após isso [5, 3].

3.2.3 Yahoo Message Broker

Embora não faça parte da especificação do SGBD, o Yahoo Message Broker (YMB) é essencial para o funcionamento do mesmo, sendo o responsável pela comunicação entre regiões [3]. É um sistema baseado em publicação-subscrição [3] [18]. Cada *tablet* é considerado um tópico e cada região subscreve ao tópico. Uma operação de atualização no PNUTS é considerada concluída com êxito após a mesma ser enviada para o YMB.

O YMB mantem um log com cada operação de atualizações feitas numa dada entrada até que todas as réplicas apliquem tais atualizações.

Entre as características do YMB relevantes para PNUTS, a mais importante é o fato de que o sistema garante que ele não perde dados persistidos nele.

3.3 Lidando com CAP

As técnicas utilizadas pelo PNUTS mais relevantes para CAP são descritas a seguir.

3.3.1 Réplica mestre

Como dito anteriormente, PNUTS além de garantir consistência eventual, também garante que a ordem das operações de atualização num dado registro é

obedecida. Para cumprir tal restrição, cada registro possui uma réplica mestre (*master*) e, em seu metadado, a informação da região onde se encontra a réplica mestre.

A réplica mestre é responsável por executar as atualizações, ou seja, quando uma requisição de atualização chega para uma réplica que não seja mestre, a mesma é redirecionada para a réplica mestre correspondente daquele registro. Após a réplica mestre concluir com êxito o comando de atualização, este é propagado assincronamente (utilizando o YMB) para as demais replicas. Como apenas a réplica principal executa atualizações, esta aplica as atualizações numa dada ordem e como todas as outras réplicas a seguem, a ordem nas atualizações é mantida. Obviamente, como a propagação das atualizações é assíncrona, num dado momento, réplicas distintas podem ter aplicado um número diferente de atualizações e consequentemente estarem obsoletas e não consistentes entre si [3, 5].

3.3.2 Contornando dados obsoletos e latência

Para permitir que usuários contornem o problema de dados obsoletos, cada registro guarda no metadado dois números, um representando quando o registro foi gerado, e o outro representando a versão do registro (uma simples contagem de inserções e atualizações). Através desses números, a API disponibiliza diferentes tipos de requisições de leitura e escrita, como por exemplo [3, 5]:

1 : Leitura simples;

2: Leitura de uma versão do dado que seja tão nova quanto ou mais recente que versão específica ;

3: Leitura da versão mais recente de uma dada entrada;

4: Escrita simples; e

5: Escrita do dado apenas se a versão atual da entrada no banco é igual ou inferior a um valor passado como parâmetro.

Os três primeiros casos estão ordenados de forma crescente, considerando-se a latência. No primeiro cenário, quando a requisição chega a uma região, esta

responde com qualquer entrada que possuir; no segundo caso, se a região não possuir uma versão tão recente quanto a desejada, a requisição é encaminhada à região que possui a réplica mestre; já no terceiro caso, a requisição é sempre encaminhada para a réplica mestre, uma vez que a mesma necessariamente possui a versão mais atual da entrada.

O sistema utiliza-se do fato de que as requisições a um dado registro possuem grande localidade espacial para tentar diminuir a latência. O sistema guarda no metadado do registro a origem das últimas (três) atualizações e , quando vantajoso, muda à localização da réplica mestre.

3.3.3 Partição na rede

Um controlador de *tablet* identifica a falha numa réplica quando esta não responde a uma requisição Quando a falha é identificada o sistema toma uma das duas providencias [3, 5]:

1. Caso a réplica falha não seja uma réplica mestre, o sistema cria mais uma réplica no *datacenter* onde a falha ocorreu, os dados da nova replica são copiados de outro centro de dados. Enquanto o processo de copia é executado, as requisições que seriam feitas à réplica falha, são encaminhadas para uma sadia. Assim, não há prejuízo para o aspecto de disponibilidade.
2. Caso a réplica falha seja uma réplica mestre, as requisições de escrita são bloqueadas para aquele objeto (afetando a disponibilidade), o sistema elege uma nova réplica para ser a réplica mestre, o sistema requisita o YMB para aplicar as atualizações necessárias na replica eleita. Ao finalizar as operações de atualização, o sistema desbloqueia as requisições de escrita.

3.4. Outras considerações

Alguns aspectos da descrição mais recente das capacidades do PNUTS que influenciam às características CAP, não foram abordados na seção anterior, pois

apresentam variações pequenas nos processos descritos ou utilizam conceitos que estão mais bem desenvolvidos em outros sistemas, e serão explicados nos capítulos seguintes. Por questões de completude esses aspectos serão citados abaixo [5].

O sistema atualmente dá suporte a outro tipo de tabela, similar à implementada pelo Dynamo, que permite escritas com nível de consistência mais relaxada, porém o sistema de resolução de conflito é pouco desenvolvido e pode causar problemas se utilizado sem cautela (é atrelado um *timestamp* a cada operação de escrita, então, a mais recente é considerada a correta, as demais são descartadas).

Outro aspecto não apresentado é que foi adicionada uma funcionalidade ao controle de réplicas do PNUTS. O sistema é capaz de criar replicas diferenciadas de entradas dentro de uma tabela. Essas entradas não precisam ser atualizadas quando uma requisição de escrita é propagada pelo sistema, pois elas não guardam os valores atrelados à chave; a entrada guarda apenas a localização das replicas que possuem tal informação. Dessa forma qualquer requisição que chegue a este tipo de réplica é simplesmente encaminhada para uma réplica normal.

3.5 Considerações finais

Como visto nesse capitulo, o PNUTS apresenta um conceito de consistência único, ele garante que toda operação de escrita sempre siga uma mesma ordem o sistema também disponibiliza mecanismos para efetuar leituras com níveis de consistência diferentes. O sistema que será analisado no próximo capítulo contrasta com o PNUTS visto que ele não permite níveis diferentes de consistência, mas informa ao cliente quando os dados estão inconsistentes e passa a responsabilidade de resolvê-las para o cliente.

CAPÍTULO 4: DYNAMO

Este capítulo explora o sistema Dynamo [10]. Inicialmente são discutidos aspectos gerais do sistema e sua arquitetura, posteriormente suas técnicas que influenciam as características CAP são analisadas e explicadas, e em seguida é apresentado uma variação do comportamento do sistema e uma melhoria possível sobre o mesmo.

4.1 Visão Geral

Sistema desenvolvido pela Amazon [10] para ser extremamente escalável, disponível e com um tempo de resposta curto. Uma de suas peculiaridades é que foi desenvolvido visando altíssima disponibilidade em operações de escritas. Os dados do sistema possuem consistência eventual.

4.2 Arquitetura

Os aspectos da arquitetura do Dynamo que serão usadas para explicar seu funcionamento são seu modelo de dados e a utilização de nós virtuais.

4.2.1 Modelo de Dados

Os dados são armazenados em tabelas, nas quais cada entrada possui apenas chave e valor. Os dados das tabelas são replicados e particionados através de uma técnica chamada *hash* consistente (*consistenthashing)* [21]. Além disso, a cada entrada é atrelado um *vectorclock* [4], que será explicado nas seções subsequentes.

4.2.2 Nós virtuais

Embora o conceito de virtualização seja comum em sistemas nas nuvens, ela está destacada aqui, pois no caso do Dynamo, além de ajudar a lidar com a heterogeneidade entre os componentes físicos utilizados pelo sistema, ela afeta algumas características relevantes em relação à CAP quando consideradas em conjunto com a técnica descrita na Seção 4.2.3.

Uma unidade de armazenamento é considerada um nó físico e pode possuir N nós virtuais dependendo de suas características físicas, como taxa de transferência e capacidade.

4.3 Lidando com CAP

A seguir serão analisadas as técnicas utilizadas pelo sistema. As duas técnicas que tem mais influencia sobre suas características CAP são a forma que ele utiliza o conceito de réplicas e o uso de *vectorclock* para fazer versionamento dos dados [10].

4.3.1 Particionamento

Dynamo particiona suas tabelas utilizando uma técnica chamada *hash* consistente [21] sobre as chaves das entradas. A saída dos valores da função *hash* dessa técnica pode ser tratada como um espaço circular ou anel. A cada nó virtual do sistema é atribuído um valor aleatório dentro deste espaço que representa a sua posição no anel. Uma entrada é atribuída a um nó da seguinte forma: a saída da função de *hash* aplicada sobre uma chave representa a posição da mesma no anel, a entrada é atribuída ao primeiro nó virtual com posição superior ao da chave (uma posição é superior à outra considerando suas posições no anel no sentido horário). Desta forma um dado nó virtual se torna responsável por todas as chaves que estão entre ele e seu antecessor.

Essa abordagem traz as seguintes vantagens [10, 21]:

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

- Se um nó não estiver disponível por um motivo qualquer, a carga é distribuída uniformemente entre os demais nós e

- Quando um nó torna-se disponível de novo ou um novo nó é adicionado ao sistema, o nó em questão passa a receber uma carga de requisições equivalente a dos outros nós.

4.3.2 Réplicas

Cada entrada é replicada N vezes. Esse processo segue da seguinte forma: após uma chave ser atribuída a um nó, esse nó passa a ser considerado o nó coordenador dessa chave. O nó coordenador é responsável por replicar os dados da entrada nos N-1 nós que o sucedem (considerando o anel descrito na seção anterior), o conjunto de nós formado por todos os nós que guardam os dados atrelados a uma dada chave é chamado de lista de preferência. Qualquer nó do sistema é capaz de descobrir quais nós fazem parte da lista de preferência de uma dada chave. A lógica por traz desse processo será explicada nos itens abaixo.

4.3.3 Versionamento de dados

Como o sistema é distribuído e apenas garante consistência eventual, é possível que existam múltiplas versões distintas de um mesmo objeto num dado instante. O sistema lida com isso, considerando o resultado de toda operação de escrita como uma nova versão (imutável) do objeto.

Os conflitos entre as versões são resolvidos quando uma requisição de leitura é feita. Normalmente, quando isto ocorre, o sistema consegue identificar qual versão deve ser usada e a retorna como resposta (esses casos são chamados de reconciliação sintática). Porem, existem casos nos quais as diferentes versões do objeto conflitam entre si e o sistema é incapaz de decidir qual usar. Nesse caso, são retornadas para o cliente todas as versões atuais válidas do objeto. Uma subsequente operação de escrita efetuada pelo cliente nesse mesmo objeto permite que o sistema resolva o conflito nas versões. Esse processo é chamado de reconciliação semântica. O sistema utiliza-se de *vectorclock* para distinguir as diferentes versões dos dados.

Como dito anteriormente na Seção 4.2.1, cada entrada num dado nó possui um *vectorclock* atrelado à mesma. Um *vectorclock* é uma lista de tuplas contendo um número representando um nó e outro representando um contador de modificações feitas nos dados [4]. Quando uma operação de escrita é efetuada, o nó que efetuou a operação altera seu *vectorclock* e o propaga para os demais. Com

isso, o sistema consegue identificar se uma versão é mais antiga que outra e se ela pode ser simplesmente sobrescrita. Para exemplificar seu funcionamento, segue abaixo o comportamento do sistema dados os seguintes eventos [10]:

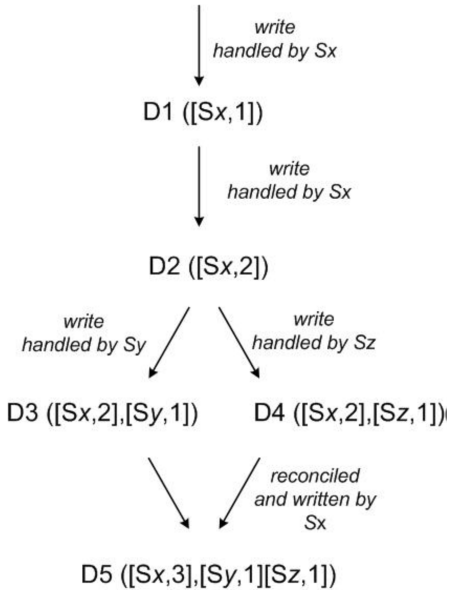
1. Um cliente faz uma requisição de escrita para uma nova entrada X. O nó Sx, que recebe a requisição, é um dos responsáveis pela chave atrelada a tal objeto. Após efetuar a requisição, ele incrementa um contador interno (exclusivo ao dado nó) e utiliza esse valor para criar o *vectorclock* da entrada. O nó possui agora uma versão D1 para tal entrada com o vetor [(Sx, 1)] associado à mesma. D1 é propagado junto com seu vetor para as demais réplicas;

2. O nó Sx recebe uma nova requisição de atualização para X, então ele considera o vetor atual [(Sx , 1)] , e atualiza a versão D1 e incrementa o vectorclock. O sistema passa a possuir a versão D2 de X e o vetor [(Sx , 2)]. Uma requisição de leitura nesse momento retornaria apenas a versão D2;

3. Duas novas requisições de escrita são feitas por clientes distintos simultaneamente. A primeira requisição chega ao nó Sy e a segunda ao Sz. Após executar as requisições, o sistema passa a possuir duas versões distintas D3 e D4 com os vetores [(Sx, 2), (Sy, 1)] e [(Sx, 2), (Sz, 1)]. D3 e D4 são sucessores de D2 (pois os valores das posições do vetor de D2 são inferiores as de D3 e D4). Então, uma réplica qualquer Kx que possuía D2, ao receber a informação propagada de Sy ou Sz sobrescreve sua versão pela recebida. Todavia, D3 e D4 não são considerados sucessores um do outro e um nó que receba a informação de um, já possuindo o outro, não pode sobrescrever a informação, pois o sistema não consegue resolver esse conflito de versão; e

4. Um cliente qualquer faz uma requisição de leitura. Nesse caso tanto D3 quanto D4 serão retornados. O cliente pode então resolver o conflito entre as versões, e caso uma nova atualização seja requisitada pelo mesmo cliente, uma nova versão D5 pode ser gerada, que será sucessora de D3 e D4.

Figura 3 - Versionamento Dynamo



Fonte: DeCandia, Giuseppe, et al [10]

A Figura 3 mostra o estado dos *vectorclocks* armazenados pelos nós no cenário descrito acima.

4.3.4 Garantias variáveis de leitura e escrita

Quando uma operação de leitura ou escrita chega a um nó X vindo do cliente, o mesmo propaga a operação inicialmente para N nós sadios (que X possa alcançar, ou seja, não estejam falhos). A operação só é considerada um sucesso se pelo menos K nós efetuem com sucesso a operação. K assume um valor W para escrita e R para leitura.

Cada instância Dynamo pode possuir valores distintos para N, R e W (desde de que R +W > N e N <= número de replicas de um dado). Esses valores impactam consideravelmente nas garantias CAP. Valores maiores de R e W aumentam a consistência, enquanto valores maiores de N aumentam a tolerância a falhas.

4.3.5 Falhas simples

Falhas são detectadas quando um nó X tenta se comunicar com outro Y, mas não recebe resposta. Quando uma requisição de escrita é encaminhada para um nó que está conhecidamente falho, a requisição é enviada a um outro nó K junto com dados da localização da réplica falha Y. K então guarda a informação de escrita

numa tabela á parte, que passa a ser monitorada periodicamente por K. A cada leitura que K faz na tabela, ela tenta contatar Y e repassar as informações de escrita.

4.4. Outras considerações

A atual implementação do Dynamo dá suporte a variações no nível de consistência no nível de tabela e não apenas no banco todo [17], como descrito na seção 4.3. Não foi possível confirmar qual o processo utilizado para garantir tal comportamento, mas uma possibilidade é:

Considerando que a consistência de uma requisição de leitura ou escrita depende dos valores variáveis R e W descritos na seção 4.3.4, esses valores são utilizados pelo nó que recebe a requisição. Dessa forma, o sistema poderia permitir variações em relação à consistência se ele permitisse a configuração dos valores de W e R em cada tabela. Ainda conseguiria permitir a variação das garantias CAP para cada requisição, em tempo de execução, se ele permitisse a definição de R e W durante a requisição específica.

CAPÍTULO 5 - BIGTABLE

Este capítulo irá descrever a arquitetura do sistema BigTable [13] que foi desenvolvido pela Google para ser um sistema de banco de dados capaz de suprir demandas variáveis, desde clientes que precisam fazer acesso à um grande volume de dados mas são tolerantes a latência, à clientes onde a latência é o fator principal. Em seguida será analisado as características do BigTable que influenciam as características CAP e por fim será feita uma pequena comparação entre as técnicas descritas até a dada seção.

5.1. Arquitetura

BigTable constitui-se de servidores de tablets *(tabletservers)* e um servidor máster. Ele utiliza-se de outros dois sistemas externos, Chubby [24] e GFS [28].

5.1.1 Modelo dados

É organizado em tabelas multidimensionais. Uma entrada na tabela é identificada por uma chave (que indica a linha), uma família de colunas, uma coluna e um *timestamp*. Toda coluna está atrelada a uma família. Todas as linhas de uma tabela possuem as mesmas famílias de colunas, porem, as colunas são flexíveis e podem variar. Dentro de uma posição, referente a uma linha específica e uma coluna, pode existir n versões distintas do item, cada uma com um timestamp atrelado ao mesmo.

A chave é representada por uma cadeia de caracteres, e as mesmas são ordenadas alfabeticamente na tabela.

A Figura 4 ilustra uma representação teórica em XML de uma tabela de estrutura válida de BigTable.

Figura 4 - Representação de uma tabela BigTable em XML

```
<tabela>
<Linha chave="1">
    <familia A>
        <coluna AA> <item tymestamp=2  valor="item"/> <item tymestamp=1 valor="items"/> </coluna AA>
        <coluna BB> <item tymestamp=1 valor="a"/>  </coluna BB>
    </familia A>
    <familia B>
    </familia B>
</Linha>
<Linha chave="2">
    <familia A>
        <coluna J> <item tymestamp=2  valor="apartamento"/> <item tymestamp=1 valor="casa"/> </coluna J>
    </familia A>
    <familia B>
        <coluna K>  </coluna K>
    </familia B>
</Linha>
</tabela>
```

Fonte: O Autor

5.1.2 Google File System

É um sistema de armazenamento distribuído utilizado pelo BigTable

Ele é constituído de um servidor mestre e vários *chunkservers*. Cada arquivo no Google File System (GFS) é subdividido em vários pedaços (*chuncks*), e a cada pedaço é atribuído um identificador único. O servidor mestre é responsável por mapear a localização física de cada *chunk*, bem como qual arquivo cada pedaço constitui. A fim de aumentar à tolerância a falha, cada *chunck* é replicado no mínimo três vezes.

Cada *chunckserver* manda constantemente uma mensagem ao servidor mestre para indicar que continua funcionando. Caso essa mensagem não chegue após certo período de tempo, ou caso um servidor não responda a uma mensagem do mestre, este considera que houve falha e todas as réplicas dos pedaços de arquivo naquele servidor falho estão inacessíveis. Caso o número de réplicas caia abaixo de um limite, o servidor mestre fica encarregado de criar uma nova réplica.

Para executar uma escrita, o cliente precisa consultar o servidor mestre. Este não permite acesso de escrita concorrente a um mesmo pedaço de arquivo. Ele também bloqueia acesso à leitura enquanto houver uma escrita pendente. Caso a

permissão para escrita seja concedida, a requisição é passada para o servidor que possui o pedaço do arquivo requisitado. O servidor então propaga a requisição para as demais réplicas, e apenas finaliza a execução do processo caso todas as réplicas tenham sinalizado o recebimento da requisição [28].

5.1.3 Chubby

Chubby é um sistema persistente e distribuído de alta disponibilidade. Ele utiliza um algoritmo da família Paxos [11, 32] para manter suas réplicas persistentes, mesmo num ambiente onde falhas na rede ocorram. Embora BigTable não implemente um algoritmo paxos diretamente, ele é uma parte fundamental de seu funcionamento, pois confere ao BigTable sua grande garantia de consistência.

5.2 Lidando com CAP

É descrito abaixo o funcionamento das técnicas utilizadas pelo sistema para tratar da restrição sobre CAP, a politica de replica e versionamento que o sistema usa também será analisada mesmo influenciando pouco as características CAP.

5.2.1 Replica e Versionamento

Olhando apenas para o contexto de um *datacenter*, o BigTable não utiliza nenhuma técnica de réplica diretamente, embora a técnica seja essencial no GFS e no Chubby. O BigTable pode ser configurado para possuir réplicas em outros *datacenters.* As réplicas se comunicam assincronamente e passam a ter consistência eventual.

Versionamento é utilizado de forma simples e não tem efeito sobre o CAP. Uma dada posição na tabela (linha, família de coluna, coluna) pode possuir N versões do dado, cada uma com um *timestamp* atrelado. Esse *timestamp* pode ser fornecido pelo BigTable ou pelo cliente. Como o sistema bloqueia acesso de escrita à linha inteira, caso ela esteja num processo de escrita, inconsistências não são criadas, existe, todavia, a possibilidade do próprio cliente fornecer o *timestamp* que deve ser atrelado ao dado. Obviamente, nesse caso o sistema não pode garantir que o dado escrito não vá colidir com versões anteriores.

O sistema descarta versões do dado utilizando um de dois critérios [13]:

1. Ele descarta a versão mais antiga do dado caso a quantidade de versões exceda um limite definido pelo cliente; ou
2. Ele descarta as versões dos dados que sejam mais antigas que um certo tempo. Esse valor também é escolhido pelo cliente.

5.2.2 Particionamento dos dados

As tabelas são particionadas horizontalmente considerando a chave, mas também podem ser particionadas verticalmente considerando as famílias de colunas.

O sistema possui dois tipos de tabelas que são tratadas de formas especiais. São diferenciadas as tabelas de metadados que guardam as localizações das partições, e uma tabela raiz, que guarda a localização das tabelas de metadado. A tabela raiz não é particionada, para evitar excesso de acessos e diminuir a latência na descoberta das posições das tabelas de metadados.

5.2.3 Coordenação dos nós

O sistema Chubby é utilizado para detecção de falhas e uniformização de carga do sistema. Todo nó que compõe o BigTable, quando funcionando normalmente, possui um *lock* (permissão exclusiva de acesso) de um arquivo especifico no Chubby. O servidor mestre monitora tais arquivos para saber quais servidores continuam ativos. Com essa informação, o servidor mestre pode verificar quais *tablets* cada nó possui. Então, o mestre pode fazer novas atribuições de *tablets* de forma a uniformizar a carga do sistema, bem como atribuir um *tablet* a um novo servidor caso quem o possui venha a falhar [13].

5.3 Comparações entre o que foi analisado

O estudo do funcionamento das abordagens empregadas pelos sistemas PNUTS, Dynamo e BigTable proporciona o conhecimento das principais técnicas de SGBDN que influenciam nas características CAP, e permite um entendimento mais fácil de outros sistemas de BDN. Um exemplo disso é o sistema Apache Cassandra [2] que foi inspirado pelo sistema BigTable e Dynamo. Ele praticamente utiliza

apenas técnicas (ligeiramente modificadas) das descritas no capítulo 4 e seções 5.2. Mesmo assim ele consegue variar seu nível de consistência em tempo de execução. O sistema expõe ao cliente cinco tipos diferentes de escrita, indo desde consistência extremamente relaxada, mas extremamente disponível, até o outro extremo. No caso da escrita com consistência relaxada, mesmo se todas as réplicas que contem a chave estiverem falhas, a escrita pode acontecer. E no caso da escrita com alta consistência, ela só finaliza após escrever em todas as réplicas. Teoricamente essa seria uma melhoria simples de se adicionar no sistema Dynamo.

Para deixar mais claras as diferenças e vantagens de cada sistema e técnica, essa seção faz comparações entre o que já foi analisado.

5.3.1 Comparação entre sistemas

Antes de comparar cada técnica, será feito uma comparação das políticas gerais de cada sistema afim de entender melhor o impacto de cada técnica.

O sistema BigTable se diferencia dos demais por garantir um nível de consistência alto. Outra peculiaridade é que ele não permite nem exige que o cliente tome ações para lidar com as características CAP, enquanto o PNUTS permite que o usuário defina níveis de consistências distintos enquanto opera o sistema, Dynamo exige que o próprio cliente resolva alguns conflitos específicos e o Cassandra permite que o usuário defina níveis variáveis de CAP além de exigir que o mesmo tome uma ação quando conflitos específicos surgem no sistema [2, 3, 10, 13].

Os sistemas Cassandra e Dynamo possuem uma arquitetura na qual todos os nós do sistema exercem a mesma função e são independentes entre si, dessa forma o sistema se torna bem mais tolerante a falhas, pois não existe uma máquina especifica que tornaria o sistema indisponível caso ela falhasse, por outro lado, uma arquitetura onde existem nós especializados pode permitir a simplificação de outros fatores do sistema. Por exemplo, o sistema PNUTS utiliza-se do conceito de réplicas mestre para garantir que sempre existe uma réplica com a versão mais recente dos dados, isso permite uma diminuição na complexidade da técnica de resolução de conflitos [2, 3, 10, 13].

A política para tratar escritas em paralelo empregada pelo BigTable influencia negativamente o aspecto de disponibilidade do sistema tanto na escrita quanto na leitura; por outro lado, os sistemas Dynamo e Cassandra, que optam por tentar sempre permitir escrita em paralelo, criam um efeito colateral de criarem versões distintas dos dados. Já o sistema PNUTS encontra um meio termo, quando há requisições em paralelo de escrita, as operações de escritas são bloqueadas, mas as operações de leitura que estão acontecendo em paralelo não; isso acaba por expor o cliente a problemas tanto de consistência quanto de disponibilidade, porém esses problemas não são tão intensos quanto os apresentados pelos outros sistemas. A Fig. 5 destaca as características que foram discutidas nessa subseção [2, 3, 10, 13].

Na subseção seguinte serão comparadas as técnicas específicas utilizadas pelos sistemas.

Figura 5 – Características dos sistemas vistos

	PNUTS	Dynamo	BigTable	Cassandra
Politica de consistência	eventual , variável para leitura	eventual	garante	variável
Particionamento	por intervalo	hash consistente	por intervalo	hash consistente
Versionamento	contador	vectorclock	timestamp	timestamp
Replicação	assincrona	assincrona	síncrona	assincrona
Otimiza operações de :	leitura	escrita	escrita	variável
Controle de escrita em paralelo	versionamento	versionamento	bloqueio de acesso aos dados	versionamento
Ponto critíco quanto a falha	réplicas mestres	não possui	tabelas raiz	não possui

Fonte: O Autor

5.3.2 Comparação das técnicas vistas

As técnicas de criação de réplicas de cada sistema são difíceis de serem analisadas individualmente, fora do contexto do sistema em que estão inseridas, pois elas podem ter efeitos negativos ou positivos em características distintas do CAP, dependendo das outras politicas e técnicas do sistema, como exemplificado na seção 2.2.3. Dessa forma, suas influencias sobre CAP não são transferíveis para outros sistemas caso elas possuam características divergentes das do sistema original, por isso essa subseção não irá comparar diretamente essas técnicas.

A técnica mais complexa de diferenciação de versões de dados vista foi a do *vectorclock*, uma vantagem clara que esta técnica possui sobre as outras, é que esta técnica de fato captura unicamente um objeto, quanto que a técnica de se utilizar apenas um *timestamp* ou um contador de operações de escrita podem identificar objetos distintos como sendo iguais caso duas requisições cheguem a replicas distintas de um sistema ao mesmo tempo. Porem, caso o sistema exponha o *vectorclock* ao cliente, como o Dynamo faz, a complexidade da resolução de conflito é repassada para o cliente, outra desvantagem é que o espaço necessário para armazenar os *vectorclock* é consideravelmente maior do que o necessário para armazenar um contador de atualizações [2, 3, 10, 13].

Outro tipo de técnica importante quanto CAP é o método de resolução de conflitos sobre versões. A técnica que o BigTale utiliza para resolver conflitos pode ser considerada preventiva, pois ele não lida diretamente com esse problema, ele utiliza a política de bloqueio de escrita de dados paralelas descritas na subseção anterior, em conjunto com o sistema GFS. O GFS por sua vez utiliza Paxos para corrigir automaticamente versões divergentes dos dados persistidos, sem expor essas divergências para o BigTable. A desvantagem dessa técnica é que o sistema se torna inflexível quanto a CAP. A técnica de resolução de conflitos do PNUTS contrasta fortemente com a do Dynamo. Enquanto a do PNUTS é extremamente simples, bastando sobrescrever a versão antiga de um dado por uma mais recente (visto que o sistema sempre sabe qual é a mais recente, pois só há um ponto onde há operações de escrita), a do Dynamo é complexa e pode exigir ação do cliente, por outro lado a técnica do dínamo aumenta consideravelmente o aspecto de disponibilidade [2, 3, 10, 13].

A técnica de particionamento de dados empregada pelo PNUTS e BigTable são similares, ambas particionam os dados em intervalos de valores das chaves, dessa forma, a proximidade dos valores de duas chaves costuma condizer com a proximidade dos nós nas quais elas estão armazenadas, o sistema BigTable utiliza esse fato para otimizar consultas a valores de dados em sequencias. Essa técnica normalmente requer um nó especializado em manter os valores dos intervalos das partições o que pode implicar numa diminuição no aspecto de disponibilidade. Em

contrapartida, á técnica utilizada pelo sistema Cassandra e Dynamo é a de *hash* consistente, ela não preserva a característica de localidade descrita na técnica anterior, mas facilita a escalabilidade do sistema (a adição de um nó afeta poucos outros) [2, 3, 10, 13].

CAPÍTULO 6. CONCLUSÃO E TRABALHOS FUTUROS

A análise realizada mostra claramente que ainda não existe uma solução de SGBDN que se destaca em relação às caraterísticas CAP. Toda política que afeta positivamente um dos seus três aspectos tem um efeito negativo em algum dos outros dois, ou na latência do sistema. A alteração de apenas uma técnica pode trazer grandes efeitos (positivos ou negativos) ao sistema como um todo.

Outro aspecto observado é que essa é uma área de intenso desenvolvimento, na qual lições e técnicas utilizadas por um sistema podem ser aproveitadas em outro. Uma classe especialmente interessante de BDN são os que compartilham parte da responsabilidade CAP com o cliente, como os sistemas que permitem clientes escolher o nível de consistência desejado em operações específicas e os que exigem que o próprio cliente resolva certos conflitos.

Trabalhos futuros envolvem analisar os esforços que os SGBDR convencionais estão fazendo para aumentar sua escalabilidade, pois eles não foram explorados neste texto.

Considerando a complexidade dos sistemas, para se ter uma clareza melhor do impacto das variações nas politicas adotadas por cada BDN, é necessário medir o desempenho de cada aspecto num cenário real de caixa branca, implantando num mesmo *datacenter* as diferentes abordagens e realizando teste de desempenho. Para esse fim, as implementações *opensource* Apache Cassandra e o Hadoop precisam ser usadas.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Abouzeid, A., Bajda-Pawlikowski, K., “HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads“. 2009.

[2] Avinash Lakshman , “Cassandra - A Decentralized Structured Storage System”

[3] A. Silberstein, J. Chen, D. Lomax, B. McMillan, M. Mortazavi, P. Narayan,R. Ramakrishnan, and R. Sears. “Pnuts in flight: Web-scale data serving at yahoo. Internet Computing”, 2012.

[4] Baldoni, R., & Raynal, M. “Fundamentals of distributed computing: A practical tour of vector clock systems.” 2002.

[5] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. “Pnuts Yahoo!’s hosted data serving platform”. Proceedings of the VLDB Endowment, 2008.

[6] Cheng, Chun-Hung, Wing-Kin Lee, and Kam-Fai Wong. "A genetic algorithm-based clustering approach for database partitioning." 2002.

[7] Codd, E. F. “A relational model of data for large shared data banks. In *Pioneers and Their Contributions to Software Engineering* “, 2001.

[8] Curino, Carlo, et al. "Schism: a workload-driven approach to database replication and partitioning." 2010.

[9] Dean, J., & Ghemawat, S. “MapReduce: simplified data processing on large clusters.” Communications of the ACM, 2008

[10] DeCandia, Giuseppe, et al. "Dynamo: amazon's highly available key-value store.": 2007.

[11] De Prisco, Roberto, Butler Lampson, and Nancy Lynch. “Revisiting the Paxos algorithm.” Springer Berlin Heidelberg, 1997.

[12] Eric Brewer,. “CAP Twelve Years Later: How the “Rules” Have Changed”

[13] Fay Chang, Jeffrey Dean, “Bigtable: A Distributed Storage System for Structured Data”

[14] Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." 2002.

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

[16] “http://wiki.apache.org/hadoop/Hbase/PNUTS” acessado em 1/4/2013.

[17] “http://aws.amazon.com/dynamodb/” acessado em 10/04/2013.

[18] Hunt, Patrick, et al. "ZooKeeper: wait-free coordination for internet-scale systems." 2010.

[19] J. L. J. Chris Anderson and N. Slater. “CouchDB The Denitive Guide”, 2010.

[20] Johansson, J. M. “On the impact of network latency on distributed systems design. Information Technology and Management”, 2000.

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

[22] Kundra, Vivek, "Federal cloud computing strategy." 2011.

[23] Liu, Yimeng, Yizhi Wang, and Yi Jin. "Research on the improvement of MongoDB Auto-Sharding in cloud environment." 2012.

[24] Mike Burrows , “The Chubby lock service for loosely-coupled distributed systems”

[25] Müller, Stephan. "The CAP-Theorem & Yahoo’s PNUTS." ,2012.

[26] Pritchett, D. “Base: An acid alternative. Queue”, 2008.

[27] Raghu Ramakrishnan , “CAP and Cloud Data Management”

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak .“The Google File System”

[29] Santhosh Kumar Gajendran , “A Survey on NoSQL Databases”

[30] Strauch, C., Sites, U. L. S., & Kriha, W.. “NoSQL databases”. 2011.

[31] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S.,. “Hive: a warehousing solution over a map-reduce framework”. Proceedings of the VLDB Endowment, 2009

[32] Tushar Chandra Robert Griesemer Joshua Redstone “Paxos Made Live - An Engineering Perspective” , 2007