pH1: middleware transacional para NoSQL

Fábio Coelho, Francisco Cruz, José Pereira, Ricardo Vilaça, and Rui Oliveira

HASLab - High-Assurance Software Laboratory
INESC TEC and Universidade do Minho
Braga, Portugal
pg18776@alunos.uminho.pt
{fmcruz,jop,rmvilaca,rco}@di.uminho.pt

Resumo As bases de dados NoSQL optam por não oferecer importantes abstracções tradicionalmente encontradas nas bases de dados relacionais, de modo a atingir elevada escalabilidade e disponibilidade: garantias transacionais e critérios de coerência de dados fortes. Estas limitações resultam em maior complexidade no desenvolvimento de aplicações sobre bases de dados NoSQL e, logo, são um obstáculo à adoção do paradigma. Neste trabalho, propomos uma camada de middleware sobre bases de dados NoSQL que oferece garantias transacionais com *Snapshot Isolation*. A abordagem é não-intrusiva, apresentando aos clientes a mesma interface NoSQL acrescida do contexto transacional. Este contexto transacional é o cerne da nossa contribuição e assenta modularmente num repositório distribuído, não persistente, que mantém várias versões dos dados manipulados e num certificador de transações concorrentes.
Apresentamos uma implementação do nosso sistema pH1 sobre Cassandra e, recorrendo a um benchmark (YCSB) extensamente utilizado na avaliação de desempenho de bases de dados NoSQL, medimos o custo do suporte do paradigma transacional com garantias transacionais ACID, que não resulta numa diminuição significativa da latência das operações quando comparado com o Cassandra.

1 Introdução

Atualmente, assiste-se a uma mudança de paradigma computacional, passando-se de aplicações locais para serviços remotos, acessíveis ubiquamente. Genericamente, este paradigma é intitulado de computação na nuvem e é concretizado recorrendo a uma abstração da sua implementação. Esta na prática é um sistema distribuído, elástico e de elevada disponibilidade, composto por grandes quantidades de máquinas oferecendo a ilusão de infinidade de recursos disponíveis [1].

O novo paradigma pressupõe milhões de utilizadores e o volume de dados gerado atingiu níveis inauditos[13]. Contudo, as bases de dados tradicionais, vulgo RDBMS (Relational Database Management System), falharam em suportar os novos requisitos essenciais de elevada escalabilidade e disponibilidade pois baseiam-se numa arquitetura altamente centralizada e rígida.

Como resposta surgiu uma nova classe de sistemas de armazenamento de dados normalmente intituladas de NoSQL. Estas bases de dados assentam numa

arquitetura distribuída, altamente escalável e disponível. No entanto, de modo a atingir elevada escalabilidade e disponibilidade estas bases de dados optam por não oferecer importantes abstrações tradicionalmente encontradas nas bases de dados relacionais que as impedem de escalar: garantias transacionais e critérios de coerência de dados fortes. Estas limitações resultam em maior complexidade e dificuldade no desenvolvimento de aplicações sobre bases de dados NoSQL, limitando o seu uso quando são necessárias garantias de coerência forte de dados.

As soluções para o modelo relacional dispõem já de soluções baseadas na existência de objetos multi-versão, nomeadamente através do nível de isolamento *Snapshot Isolation*. Este nível de isolamento faz com que transações concorrentes trabalhem sobre diferentes versões, podendo aumentar o nível de concorrência quando comparado com sistemas baseados em exclusão mútua.

Neste artigo propomos uma camada de *middleware* sobre bases de dados NoSQL que oferece garantias transacionais recorrendo a uma implementação baseada em *Snapshot Isolation*. A abordagem seguida é não-intrusiva baseando-se inteiramente na interface de cliente e nas garantias disponibilizadas pela base de dados NoSQL subjacente. Esta camada apresenta aos clientes a mesma interface NoSQL acrescida do contexto transacional.

O contexto transacional é o cerne da nossa contribuição e assenta modularmente num repositório distribuído, não persistente, que mantém várias versões dos dados manipulados e num certificador responsável pelo controlo multi-versão de transações concorrentes.

Com vista a aferir o custo da introdução de garantias transacionais, conduzimos um conjunto de testes que visaram avaliar este custo em função do desempenho prévio da base de dados NoSQL.

Este artigo está organizado da seguinte forma. As secções 2 e 3 oferecem uma visão global dos conceitos abordados. A secção 4 descreve a arquitetura e detalha a construção dos principais componentes, enquanto que a secção 5 descreve a sua implementação. A secção 6 apresenta a avaliação do sistema. A secção 7 aborda trabalho relacionado e a secção 8 conclui o trabalho.

2 Sistemas Transacionais

Uma transação é vista como uma unidade de trabalho que define uma computação completa e correta, podendo ser composta por diversas operações singulares de escrita/leitura. Agrupar operações numa transação introduz um nível de independência de tratamento em relação a outras transações. O principal objetivo é garantir que todas as operações contidas numa transação sejam executadas atomicamente, devendo ser iniciada e terminada com um estado coerente dos dados.

Um sistema transacional, tem de garantir que as propriedades `ACID`[1] ocorram como um todo; por forma a assegurar a atomicidade, coerência, visibilidade e

[1] Atomicity, Consistency, Isolation, Durability

persistência de todas as operações contidas no âmbito de uma transação. Em especifico nesta contribuição, daremos ênfase ao isolamento e atomicidade; uma vez que, a coerência e durabilidade ficarão sob a responsabilidade da camada superior e inferior ao middleware.

2.1 Snapshot Isolation

O uso de transações surge associado a diferentes necessidades de isolamento, cujo nível mais elevado é *Serializability*, no qual há equivalência a uma execução sequencial [4] - uma execução sem concorrência.

Snapshot Isolation é um tipo de controlo de concorrência multiversão, que constrói um nível de isolamento mais relaxado [3], recorrendo a carimbos temporais por forma a que cada transação opere sobre um *snapshot* coerente dos dados [12].

Segundo este nível de isolamento, uma transação caracteriza-se por dois marcadores distintos, que definem a sua criação (ts) e, o momento em que o commit[2] da transação é autorizado (tc). Uma transação fica limitada ao intervalo definido por estes carimbos temporais. O sucesso de uma transação implica que não existam outras transações cujo período de vida se sobreponha ao da transação cessante e, que partilhem dados a ser escritos.

Os níveis de isolamento caracterizam-se por serem mais, ou menos permissivos na quantidade e tipo de incoerências que toleram. O *Snapshot Isolation* posiciona-se perto do nível mais forte (*Serializability*); contudo, apesar de evitar praticamente todas as anomalias caracterizáveis [3,12], poderá ocorrer um tipo de anomalia causada pela falta de sincronia de operações de escrita (*write skew*), que não é naturalmente resolvida. No entanto para muitas aplicações é possível correr com controlo de concorrência *Snapshot Isolation* e ainda assim obter uma execução *Serializable* [7].

2.2 Ordenação de Transações e Certificação

Num sistema concorrente, a garantia de ordem na execução de operações concorrente é assegurada por um *escalonador* ou por um *certificador*. Apesar de ambas as abordagens poderem ser baseadas em exclusão mútua ou carimbos temporais; ao recorrer a carimbos, as garantias de ordem total ficam imediatamente satisfeitas. Tal é verdade uma vez que a emissão de carimbos temporais assegura que o carimbo B é necessariamente maior que o carimbo A, caso $A \rightarrow B$.

Duas transações, caracterizadas por, $[ts_A, tc_A]$ e $[ts_B, tc_B]$ são concorrentes caso $ts_A \geq ts_B$ ou $ts_B \geq ts_A$ e $ts_A < tc_B$ ou $ts_b \leq tc_A$. O processo de certificação será responsável por assegurar que para duas transações concorrentes, os tuplos que cada uma manipula não se sobrepõem.

[2] O commit de uma transação refere-se ao momento em que o certificador permite que a transação seja terminada; por não introduzir incoerências no sistema.

3 Cassandra

O Cassandra [10] é uma base de dados NoSQL, distribuída, sem suporte de versões, que permite particionar grandes volumes de dados, enquanto oferece alta disponibilidade dos mesmos. O seu modelo de dados pode ser visto como "chave / lista arbitrária de atributos". Os dados encontram-se acessíveis através de uma `API` que permite operações de inserção, leitura e remoção de dados, mas não existindo forma de executar interrogações complexas sobre estes. O particionamento dos dados pode seguir duas abordagens distintas: aleatória ou ordenada. Por omissão recorre-se ao modo aleatório que usa uma técnica de particionamento horizontal intitulada *consistent hashing* [9].

O Cassandra usa uma estratégia de replicação que depende do facto de um cluster se dispersar por vários ou apenas um centro de dados. Com um centro de dados, a estratégia de replicação empregue (Simple Strategy) considera um nível de replicação k. Quando um tuplo é inserido no sistema, em primeiro lugar é persistido no nó que o particionador escolheu como responsável, em segundo lugar, e dado que os nós mantém a noção dos seus vizinhos (i.e. arquitetura em anel), é replicado nos k nós seguintes.

Cada nó Cassandra recebe pedidos de escrita ou leitura. Quando um cliente se liga a um nó, este torna-se o coordenador para o pedido. O coordenador, dependendo da estratégia de particionamento e replicação decide quais os nós que deverão ser contactados por forma a atender o pedido. Caso seja uma escrita, o coordenador contacta todas as réplicas ativas de acordo com o particionador (independentemente do nível de coerência, que servirá apenas para ditar quantas réplicas devem responder). Caso seja um pedido de leitura, o coordenador contacta o número de réplicas imposto pelo nível de coerência e coleciona as suas respostas comparando-as por forma a detetar incoerências. Caso sejam detetadas, e dado que cada escrita tem associada um marcador temporal; o cliente receberá a resposta mais recente de entre as respostas que o coordenador recebeu. Simultaneamente, o coordenador decreta uma escrita a todas as réplicas desatualizadas (read repair request) de modo a restaurar o nível de coerência definido para o tuplo.

O Cassandra não oferece garantias transacionais `ACID`, trocando o isolamento e atomicidade por alta disponibilidade e performance na execução de escritas.

4 pH1

O pH1 é um *middleware* que se posiciona entre o cliente e a base de dados NoSQL por forma a introduzir garantias transacionais. Genericamente, este segue a `API` exportada pela base de dados NoSQL, adicionando a possibilidade de iniciar e terminar transações com garantias de isolamento e atomicidade. Para tal, é introduzida a noção de Transação (início e final), e também de Operação na qual as operações de leitura e escrita da `API` NoSQL são mapeadas.

Como descrito anteriormente, oferecer o nível de isolamento *Snapshot Isolation* implica manter múltiplas versões dos dados manipulados. De modo a fácil e

modularmente suportar várias bases de dados NoSQL, mesmo aquelas que não suportam multi-versionamento de forma nativa como é o caso do Cassandra, foi necessário introduzir um repositório de versões não persistente (RVNP), por forma a satisfazer as operações de leitura de objetos multiversão. Dessa forma a base de dados NoSQL armazenará apenas a versão mais recente de cada tuplo, enquanto que o RVNP manterá as versões anteriores. O final (*commit*) de uma transação fica sujeito às condições descritas na Secção 2.2, decisão que será tomada por um certificador externo.

O pH1 é composto por três componentes: o Gestor Transacional (GT), o Repositório de Versões Não Persistente (RVNP) e o Certificador (TSO), como fica patente na Figura 1. No protótipo atual do pH1 a base de dados NoSQL usada foi o Cassandra, doravante referido como base de dados NoSQL.

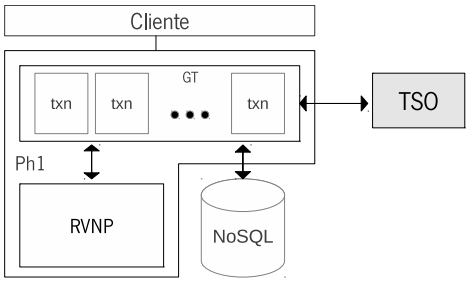

Figura 1. Arquitetura do pH1.

4.1 Gestor Transacional.

O gestor transacional é o componente central do pH1. O cliente poderá iniciar uma transação e nesse contexto efetuar um conjunto de operações tal como na `API` NoSQL. No final poderá pedir o *commit* da transação. O Gestor Transacional detém o contexto (*txn* na Figura 1) ou *snapshot* de todas as transações ativas.

O pedido de nova transação é intermediado pelo pedido ao TSO de um novo marcador de leitura para a transação (ts). No contexto de uma transação, o cliente dispõem das operações de leitura (unária ou em intervalo) e escrita de objetos, tal como na `API` NoSQL.

Operação de escrita. A operação de escrita cria a operação respetiva no *snapshot* da transação a que diz respeito. As operações que existem nesse contexto são apenas traduzidas nas camadas de dados durante a finalização (i.e. *commit*) da transação respetiva.
Operação de leitura. A operação de leitura, quer seja unitária ou em intervalo, recorre ao marcador de leitura da transação (ts) por forma a obter, de acordo com o nível de isolamento, a versão mais recente (na visão da transação) para o(s) tuplo(s) a ser(em) lido(s). A chave do tuplo é uma composição da tabela, chave e coluna(s) a ser(em) lida(s). A versão a ser lida é aquela que detém o

maior marcador temporal, menor que o marcador de leitura da transação (ts). Esta versão pode existir no *snapshot* da transação (caso resulte de uma escrita da própria transação), na base de dados NoSQL (caso resulte da última transação que modificou com sucesso o objeto) ou no RVNP (caso resulte de uma transação que modificou com sucesso no passado o objeto). As estruturas de dados são acedidas na ordem descrita, disponibilizando o primeiro objeto que corresponda à chave a ser lida e cumpra a condição de versão mais recente, de acordo com o Algoritmo 1. A operação de leitura de um intervalo (scan) apenas difere da

Algorithm 1: Procedimento de leitura de um item - get

```
Entrada: tabela, chave, coluna
chave pesquisa ← tabela + chave + coluna
se chave pesquisa ∃ snapshot(escrever) então
    retorna snapshot.ler(chave pesquisa)
senão
    valor_NoSQL ← NoSQL.ler(tabela, chave, coluna)
    se valor_NoSQL ≤ marcador_leitura_transacao então
        retorna valor_NoSQL
    senão
        valor_RVNP ← RVNP.ler(tabela, chave, coluna)
        retorna valor_RVNP
```

versão unária na medida em que são procuradas as versões mais recentes para o conjunto de chaves e colunas solicitadas.

Operação de *commit*. A operação de *commit* é invocada no final da transação, traduzindo as operações que existem no contexto da transação nas camadas de dados. Esta operação está restrita ao gestor transacional e apenas ocorre quando o cliente pede o *commit* da transação.

A resposta positiva do certificador tem por base as modificações da transação a ser certificada e das transações concorrentes. A resposta positiva vem acrescida do marcador de escrita para a transação. As versões antigas, para cada linha, que no contexto desta transação residiam na base de dados NoSQL são agora escritas no RVNP e as novas versões (que até agora constavam apenas no *snapshot* da transação) são escritas no base de dados, de acordo com o Algoritmo 2.

Note-se que o processo de escrita pode resultar numa falha e especificamente no caso do Cassandra pode dar origem a dados incoerentes. Nesse caso, as operações de leitura subsequentes são obrigadas a verificar tanto o Cassandra como o RVNP na busca da versão mais recente para um tuplo, em vez de retornarem o primeiro tuplo que cumprisse a condição descrita na secção 4.1. Tal comportamento era possível, já que na ausência de falhas a versão mais recente está sempre em primeiro lugar no *snapshot* da transação seguida da base de dados e por fim no RVNP. Se não fosse tomado este comportamento, uma transação futura poderia inferir que possuía a versão mais recente para um elemento obtido do NoSQL (uma vez que cumpre a condição descrita na Secção 4.1), quando na

Algorithm 2: Procedimento de *commit*

```
Entrada: marcador escrita
para cada tabela ∈ snapshot faça
    colunas adicionar ← snapshot(colunas a adicionar)
    valores NoSQL ← NoSQL.ler(tabela, colunas adicionar)
    para cada valor ∈ colunas adicionar faça
        se valor ∈ valores NoSQL então
            RVNP.adicionar(valor, marcador escrita)
    NoSQL.adicionar(colunas adicionar, marcador escrita)
```

realidade a versão mais recente reside no RVNP. Este comportamento só pode ocorrer no período entre a falha de escrita no Cassandra e a atuação dos seus mecanismos de recuperação.

Operação de *limpeza.* À medida que as transações terminam, deixa de ser necessário a manutenção das versões associadas no RVNP. Em cada ciclo de limpeza, o gestor transacional determina qual o marcador temporal de leitura da transação mais antiga que ainda se encontra ativa. Este marcador é passado ao RVNP que removerá as versões obsoletas. O ciclo de limpeza decorre a cada três minutos, sendo um parâmetro ajustável.

4.2 RVNP Repositório de Versões Não Persistente

A utilização do nível de isolamento descrito requereu a introdução de um Repositório com vista a manter versões anteriores de objetos de acordo com as necessidades e comportamento descritos na secção 2.1 e na secção 4.1.

Introduzimos um repositório distribuído ao longo de vários nós que exportam uma `API` que disponibiliza operações de escrita e leitura.

Os nós que compõem o repositório são completamente independentes dos nós NoSQL anteriormente referidos e comunicam entre si por via de um *toolkit* de comunicação em grupo [5]. Ao recorrerem a este *toolkit* estabelece-se a noção de grupo – cuja pertença é gerida pelo *toolkit* – criando uma via de troca de mensagens no contexto desse grupo. A utilização deste *toolkit* garante ainda a troca confiável e atómica de mensagens, reenviando-as em caso de extravio.

Qualquer um dos nós que compõem o RVNP está apto a receber pedidos de escrita e leitura de um cliente.

Operação de *escrita.* A receção de um pedido de escrita num dos nós do RVNP causa o envio de uma mensagem em *multicast* para o grupo. Todos os nós (incluindo o que enviou a mensagem) efetuam a escrita na sua visão local. No final, o nó que recebeu o pedido responde ao cliente o resultado da operação.

Operação de *leitura.* Devido à estratégia de replicação, qualquer nó está apto a responder a um pedido de leitura de um elemento. Quando um nó recebe este pedido de um cliente, recorre apenas à sua visão local para produzir uma resposta com o elemento interrogado. Durante a execução deste pedido, não existe

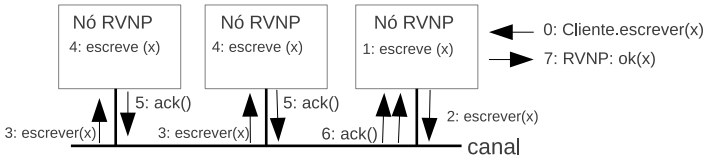

Figura 2. Operação de escrita. A numeração na figura representa a sequência de mensagens trocadas.

qualquer comunicação entre os nós do RVNP.

Operação de *limpeza*. O mecanismo de limpeza (*eviction*) ocorre a cada ciclo de limpeza do gestor transacional. O RVNP procederá então à remoção, para todas as chaves nele contidas, de todas as versões que sejam anteriores ao marcador determinado pelo gestor transacional.

4.3 Certificador e Oráculo de Marcadores Temporais

Introduziu-se um certificador e oráculo externo, com vista a gerar marcadores temporais e certificar transações, o OMID [14]. Este projeto foi desenvolvido pela Yahoo! e proporciona um certificador Snapshot Isolation (The Status Oracle) independente de uma plataforma de suporte (no seu caso o HBase [8]).

Enquanto oráculo, o TSO proporciona um método de obtenção de marcadores temporais únicos, com garantia de ordem total. O gestor dita ao oráculo o incremento do marcador no momento da criação de uma nova transação.

Como certificador, o TSO encarrega-se de verificar se existe algum conflito de escrita associado à transação que está a estudar (de acordo com a secção 2.2) e autoriza, ou não, a certificação da mesma. O certificador tem poder para tomar esta decisão uma vez que detém informação sobre o sucesso, ou não, de todas as transações. Autorizar a certificação causa o incremento do marcador temporal, que é devolvido ao gestor transacional e usado como marcador de escrita para a transação em causa.

5 Implementação

Na implementação do pH1 foi usada a linguagem orientada aos objetos Java. Recorreu-se ainda a três *toolkits* externos, por via a possibilitar a interação com o Cassandra, com o TSO e entre os vários nós do RVNP. Para o acesso ao Cassandra foi utilizada uma API popular, a *hectorAPI*. Esta API possibilitou uma via simples para comunicar com o Cassandra. A comunicação entre o gestor transacional e o TSO ficou a cargo de uma API baseada num *toolkit* de comunicação assíncrona, guiada por eventos: Netty[3].

[3] http://netty.io

A comunicação entre os vários nós RVNP foi implementada recorrendo a um *toolkit* de comunicação em grupo. Nesta implementação usámos o JGroups[5].

Como base de dados NoSQL usou-se o Cassandra por ser uma base de dados bastante popular e por não suportar nativamente multi-versionamento.

6 Avaliação

Nesta secção avaliamos o nosso sistema pH1 de modo a quantificar a penalização imposta pela introdução de garantias transacionais sobre Cassndra de forma não intrusiva. Para tal recorremos a um *benchmark* extensamente utilizado na avaliação do desempenho de bases de dados NoSQL, o YCSB. A versão usada nesta avaliação foi ligeiramente modificada de modo a incluir uma nova operação que executa várias atualizações numa única operação, como acontece numa transação tradicional. Esta operação (multiUpdate) modifica 10 chaves aleatórias em cada operação. Importa referir que cada operação singular (i.e. leituras e escritas) são tratadas como transações no caso do pH1 (a exemplo do *autocommit* do JDBC).

6.1 Configuração e testes

A execução destes destes decorreu num cenário distribuído ao longo de cinco máquinas (Intel i3-2100-3.1GHz-quadcore 64 bit, 4 Gb RAM) iguais, interligadas por uma rede Gigabit. O ambiente de teste foi o seguinte: duas máquinas formaram o cluster Cassandra, uma máquina encarregou-se apenas do certificador (TSO), enquanto que as restantes duas executaram um cliente do YCSB cada. A execução do cliente YCSB nas duas últimas máquinas instanciou em cada uma delas um Gestor Transacional (no caso do pH1).

Cada teste dividiu-se em duas fases. A primeira fase consistiu em popular o sistema com aproximadamente um milhão de entradas, cada uma com aproximadamente 1KB de tamanho. A segunda fase consistiu em executar o YCSB, dispondo 25 clientes em cada gestor, num total de 50 clientes. Contabilizaram-se 250000 operações de acordo com o rácio: 45% leituras, 12.5% updates, 12.5% multiUpdates e 30% scans.

6.2 Resultados

Os resultados que apresentamos dizem respeito à média das latências das duas instâncias YCSB, e também correspondem à média dos resultados de cinco testes independentes. Tal como expectável, a introdução das garantias transacionais faz com que o pH1 apresente um nível de desempenho inferior ao Cassandra. No geral, o pH1 apresentou uma latência por tipo de operação superior ao Cassandra (Figura 3). No entanto, para as operações de leitura, *Read* e de *Scan*, a diferença na latência é muito ténue. Já quanto às operações de actualização de dados, *Update* e *multiUpdate*, a diferença é superior. Este resultado é explicado pelo facto de o Cassandra fazer uma escrita direta em cada operação, enquanto que o pH1 ao envolver a operação no contexto transacional necessita de aguardar pela

certificação. Se esta for bem sucedida, implica ler a versão que estava persistida no Cassandra e guardá-la no RVNP, e finalmente persistir a nova versão no Cassandra. Este comportamento justifica a disparidade de latências neste tipo operações.

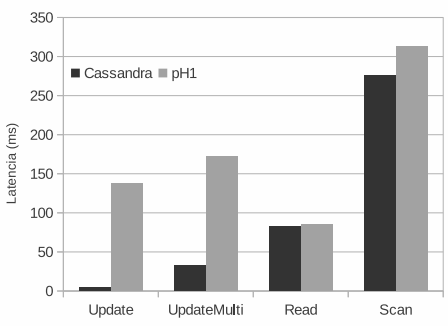

Figura 3. Avaliação do custo introduzido pelo pH1 em relação ao Cassandra.

Salienta-se ainda que os ciclos de limpeza que ocorrem no RVNP, dado a sua frequência, são desprezáveis. Já as escritas no RVNP, quando desativadas promovem uma diminuição na latência das operações de Update e MultiUpdate, na ordem dos 2 milisegundos por operação.

7 Trabalho Relacionado

Existem vários trabalhos que oferecem garantias transacionais e critérios de coerência de dados fortes sobre bases de dados NoSQL, nomeadamente:
CloudTPS [15]; MegaStore [2]; ElasTras [6]; Percolator [11]; OMID.

O sistema CloudTPS oferece garantias transacionais (ACID) sobre uma qualquer base de dados NoSQL. Para isso, introduz gestores de transações locais (LTM). Um LTM contém uma cópia de uma partição de dados armazenado na base de dados NoSQL e, de acordo com a carga no sistema, poderá haver várias instâncias de LTMs. Cada LTM é responsável por garantir a coerência da sua partição. Para transações que envolvam dados que se encontrem em diferentes LTM é usado usando um protocolo de *two-phase commit* (2PC). Os sistemas MegaStore e ElasTras são semelhantes ao anterior. Estes sistemas diferem do pH1 pela sua arquitetura e não usarem *Snapshot Isolation* como nível de isolamento.

O Percolator implementa *Snapshot Isolation* sobre o BigTable usando uma abordagem distribuída baseada em exclusão mútua. Para isso adiciona duas colunas a cada tuplo: *lock* e *marcador temporal de escrita*. O uso de exclusão mútua facilita a detecção de conflitos de escrita, mas transações de leitura têm que verificar o estado da transação que detêm *lock* o que impõe uma carga extra nos servidores e tem um grande impacto na performance.

Contrariamente ao Percolator, o OMID implementa *Snapshot Isolation* sobre HBase [8] mas recorrendo a uma abordagem que não usa primitivas de exclusão mútua. Como descrito na Secção 4, o pH1 reúsa o certificador e oráculo de marcadores temporais do OMID, eliminando todos os restantes componentes (e.g. camada persistência, clientes transacionais). A grande diferença entre a nossa abordagem e o OMID centra-se no modo como as várias versões do mesmo objeto são persistidos; enquanto que para as restantes é o facto de se distanciar do uso de primitivas de exclusão mútua como protocolos de *two phase commit* ou o uso de *locks*. No OMID as múltiplas versões são guardadas diretamente no HBase, ou seja é dependente da base de dados NoSQL subjacente. Enquanto que na nossa abordagem, as múltiplas versões do mesmo objecto são mantidas no RVNP, o que permite que esta seja genérica, não intrusiva e por isso aplicável a qualquer base de dados NoSQL, mesmo em bases de dados que não suportem múltiplas versões nativamente, como é exemplo o Cassandra.

8 Conclusão e Trabalho Futuro

Neste artigo apresentámos o pH1, uma camada de *middleware* que oferece de forma genérica e não intrusiva garantias transacionais sobre bases de dados NoSQL. A abordagem baseia-se inteiramente na interface cliente da base de dados NoSQL, mas acrescenta um contexto transacional que permite iniciar e terminar transações. As garantias transacionais ACID oferecidas pelo pH1 são baseadas no nível de isolamento *Snapshot Isolation* e suportadas por um repositório distribuído, não persistente, que mantém várias versões dos dados manipulados. Para além disso, o pH1 usa um certificador que é responsável pelo controlo multi-versão de transações concorrentes.

O protótipo desenvolvido assenta sobre a base de dados Cassandra, e recorrendo a um benchmark extensamente usado em bases de dados NoSQL (YCSB) permitiu-nos avaliar o custo do suporte do paradigma transacional, relativamente às latências das diversas operações disponíveis. Os resultados mostram que a diferença nas operações de leitura é marginal, enquanto que as operações de actualização incorrem numa maior penalização devido ao modo como as escritas se processam em contexto transacional.

Pelos resultados obtidos podemos concluir que a abordagem é válida e poderá permitir que o paradigma NoSQL seja mais facilmente adotado. Os resultados apresentados são positivos tendo em conta que o workload alvo é maioritariamente composto por leituras. Seria ainda interessante usar uma segunda implementação NoSQL por forma a provar que as mesmas garantias se mantém, e também verificar o desempenho do pH1 segundo uma avaliação mais intensiva; possivelmente recorrendo a um *benchmark* TPC.

Agradecimentos

Este trabalho foi parcialmente financiado por: FEDER - Fundo Europeu de Desenvolvimento Regional - através do programa COMPETE (Programa ope-

racional para a competitividade) e FCT - Fundação para a Ciência e Tecnologia - sob o projeto Stratus/FCOMP-01-0124-FEDER-015020; através do projeto Pest/FCOMP-01-0124-FEDER-022701 e o European Union Seventh Framework Programme (FP7) sob contrato de subvenção nº 257993 (CumuloNimbo).

Referências

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53, 50–58 (April 2010)
2. Baker, J., Bondç, C., Corbett, J.C., Furman, J.J., Khorlin, A., Larson, J., L´eon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: In Conference on Innovative Data Systems Research (CIDR). pp. 223–234 (Jan 2011)
3. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. SIGMOD Rec. 24(2) (May 1995)
4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems (1986)
5. Carvalho, N., Pereira, J., Rodrigues, L.: Towards a generic group communication service. In: Meersman, R., Tari, Z. (eds.) On The Move To Meaningful Internet Systems, International Symposium on Distributed Objects, Middleware, and Applications (DOA). Lecture Notes in Computer Science, vol. 4276, pp. 1485–1502 (2006)
6. Das, S., Agrawal, D., El Abbadi, A.: Elastras: an elastic transactional data store in the cloud. In: Proceedings of the 2009 conference on Hot topics in cloud computing. HotCloud'09, USENIX Association, Berkeley, CA, USA (2009)
7. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2), 492–528 (Jun 2005)
8. George, L.: HBase: The Definitive Guide. O'Reilly Media, 1 edn. (2011)
9. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 654–663. STOC '97, ACM, New York, NY, USA (1997)
10. Lakshman, A., Malik, P.: Cassandra - A Decentralized Structured Storage System. In: LADIS'09 (2009)
11. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (2010)
12. Revilak, S., O'Neil, P., O'Neil, E.: Precisely serializable snapshot isolation (pssi)
13. Skillicorn, D.: The case for datacentric grids. Tech. rep., Department of Computing and Information Science, Queen's University (November 2001)
14. Yabandeh, M., Gómez Ferro, D.: A critique of snapshot isolation. In: Proceedings of the 7th ACM european conference on Computer Systems. pp. 155–168. EuroSys '12, ACM, New York, NY, USA (2012)
15. Zhou, W., Pierre, G., Chi, C.H.: Cloudtps: Scalable transactions for web applications in the cloud. IEEE Transactions on Services Computing 99(PrePrints) (2011)