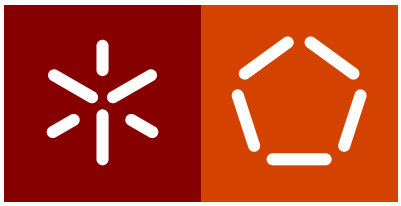
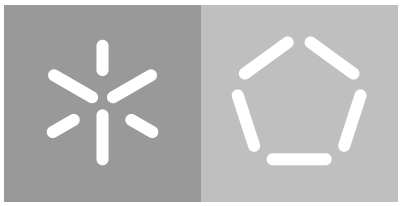


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Cláudia Fernandes Ribeiro

Middleware de acesso coerente a serviços de bases de dados na nuvem

Dezembro 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Cláudia Fernandes Ribeiro

Middleware de acesso coerente a serviços de bases de dados na nuvem

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho realizado sob orientação de
José Orlando Roque Nascimento Pereira
Ana Luísa Parreira Nunes Alonso

Dezembro 2017

AGRADECIMENTOS

Como seria impossível chegar a este marco da minha vida sozinha, gostaria de agradecer a algumas pessoas que me ajudaram das mais diversas formas.

Em primeiro lugar, aos meus orientadores: o Professor José Orlando Pereira e à Ana Nunes Alonso, pela orientação, ajuda, disponibilidade e constante passagem de conhecimento durante a elaboração desta dissertação.

A todos os meus amigos que sempre me acompanharam ao longo deste percurso.

A toda a minha família pela compreensão, pelo incentivo e motivação que sempre me deram.

Agradeço aos meus pais, que me deram a oportunidade de concretizar as minhas ambições. À minha irmã deixo um agradecimento especial.

A todos os que deram o seu contributo, direta ou indiretamente, o meu muito obrigada.

ABSTRACT

The increasing amount of data to process, in several domains, led to the need to scale storage systems. In addition to traditional database systems, that support ACID (Atomicity, Consistency, Isolation, Durability) transactions, systems based on other paradigms emerged, which have simpler operations, based on the key-value model. In these systems, support for ACID transactions was dropped in favor of scalability.

On the other hand, there are cloud storage services offering the key-value model in which the billing plan is based on the number of operations provisioned and the consistency level at which these are executed.

However, some applications still require data access with consistency guarantees. For this purpose, transactional layers that interface with key-value storage systems by intercepting all applications requests have emerged.

This dissertation contributes an analysis of trafe-offs in consistency models offered by cloud storage services and proposes an architecture which takes advantage of intercepting accesses to the cloud service to optimize performance and cost. This proposal is evaluated with a simulation model, that shows its validity.

RESUMO

O aumento da quantidade de dados a processar, em diversos domínios, levou à necessidade de escalar os sistemas de armazenamento. Além dos sistemas de bases de dados tradicionais, que têm suporte a transações com propriedades ACID (Atomicidade, Coerência, Isolamento, Durabilidade), surgiram sistemas com base em outros paradigmas, que oferecem operações mais simples, baseadas no modelo chave-valor. Nestes sistemas, abdicou-se do suporte a transações com propriedades ACID para atingir a escalabilidade necessária.

Por outro lado, apareceram os serviços de armazenamento na nuvem, seguindo o modelo chave-valor, em que o tarifário de utilização é baseado no número de operações aprovisionadas e o nível de coerência a que estas são executadas.

No entanto, continuam a haver aplicações que necessitam de aceder a dados com garantias de coerência. Para tal, surgiram camadas transacionais de interface com sistemas de armazenamento chave-valor que medeiam todos os acessos das aplicações ao serviço de armazenamento.

Esta dissertação analisa os compromissos dos modelos de coerência, oferecidos por serviços de armazenamento na nuvem, e propõe uma arquitetura que tira partido da mediação dos acessos à nuvem para otimizar o custo e o desempenho. Esta proposta é avaliada com um modelo de simulação, que permite demonstrar a sua validade.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE EXCERTOS

SIGLAS

A

ACID Atomicidade, Coerência, Isolamento, Durabilidade.

B

BD Base de Dados.

C

CAP *Consistency, Availability, Partition tolerance*.

D

DC *Data Component*.

I

ID Identificador único.

K

KB *Kilobyte*.

L

LTM *Local Transaction Manager*.

R

RU *Request Unit*.

S

SGBD Sistema de Gestão de Bases de Dados.

SI *Snapshot Isolation*.

T

TC *Transactional Component*.

U

UM Unidades Monetárias.

W

WCRT *Weakly-Consistent Read-only Transaction*.

Y

YCSB Yahoo! Cloud Serving Benchmark.

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Os dados armazenados são atualmente considerados a componente mais valiosa de um sistema de informação. As tecnologias de gestão de bases de dados, que gerem e controlam os acessos a coleções organizadas de dados, têm por isso uma papel central no projeto, concretização e implantação de sistemas informáticos.

Sendo necessário que vários utilizadores consultem e modifiquem ao mesmo tempo os dados que estas contêm, um problema evidente é o controlo do acesso concorrente a esse conjunto de informações que evite a existência de conflitos, isto é, situações em que a Base de Dados (BD) tenha incoerências devido a escritas sobre o mesmo recurso vindas de diferentes origens (Nguyen, 2002). Considerando um cenário em que dois utilizadores estão, simultaneamente, a aceder aos mesmos dados da BD, basta que um deles esteja a atualizar informação na base de dados para poder haver incoerências nos resultados obtidos.

A resolução deste problema é um dos objetivos principais de um Sistema de Gestão de Bases de Dados (SGBD) (Connolly and Begg, 2004), que endereça também outros desafios, tais como o controlo da redundância de dados, maior acessibilidade e a manutenção da integridade dos dados. As soluções propostas para estes problemas dividem os sistemas de gestão de bases de dados em dois paradigmas principais: o relacional e transacional, referido vulgarmente como SQL, e o não relacional, referido também como NoSQL.

Nos sistemas de bases de dados SQL os dados são organizados de acordo com o modelo relacional, em tabelas, e consultados de forma declarativa através de interrogações. Cabe pois aos sistemas de gestão de bases de dados determinar qual a forma ideal para executar essa consulta, aliviando assim o programador de aplicações e proporcionando a capacidade do programa se adaptar a diferentes conjuntos de dados e ambientes.

Nestes sistemas o controlo de acessos é transacional e garante as propriedades ACID. Uma transação consiste num conjunto de operações de leitura e escrita sobre um determinado SGBD (Ferro et al., 2014) que, uma vez executada, poderá ter dois resultados diferentes: **confirmada** (*committed*) ou **revertida** (*aborted*). Quando uma transação é confirmada

isso significa que foi concluída com sucesso, isto é, o resultado de todas as operações efetuadas é refletido na base de dados. Quando a transação é revertida nenhum dos resultados é aplicado à BD. Cada transação deverá respeitar um conjunto de propriedades conhecidas como ACID (Haerder and Reuter, 1983):

- **Atomicidade:** ou é refletido o resultado de todas as operações na BD ou então não é o de nenhuma;
- **Coerência:** a execução isolada da transação preserva a coerência da BD;
- **Isolamento:** os efeitos da execução concorrente de transações são equivalentes ao de uma execução em série das mesmas;
- **Durabilidade:** depois de uma transação ser aplicada, todas as alterações feitas à BD persistem.

Um pressuposto que deve ser considerado é que o resultado das transações, mesmo executadas concorrentemente, deverá ser equivalente a uma possível execução em série das mesmas. A este conceito dá-se o nome de ***Serializability*** (Bernstein et al., 1986).

Já no paradigma não relacional ou NoSQL, existem diferentes modelos de dados como, por exemplo, o modelo chave-valor, podendo os campos serem diferentes em cada entrada. A responsabilidade pela definição do plano de execução de uma interrogação fica assim nas mãos do programador de aplicações. Este paradigma abdica também das garantias transacionais com o objetivo de, mais facilmente, conseguir armazenar e processar eficientemente grandes quantidade de dados.

Este paradigma tem sido particularmente adotado na computação na nuvem, cujo principal objetivo é tornar a computação um serviço acessível ao público em geral, oferecendo recursos facilmente configuráveis como, por exemplo, servidores, armazenamento, entre outros, reduzindo o custo para os utilizadores que recorrem a este tipo de serviços e trazendo, também, melhor performance, escalabilidade, acessibilidade e disponibilidade (Kossmann et al., 2010).

Várias empresas como, por exemplo, a Amazon (Amazon), a Microsoft (Microsoft), a Google (Google), entre outras, fornecem serviços de computação e armazenamento na nuvem, podendo estes ser utilizados por diferentes aplicações, respeitando os respetivos requisitos, sendo aplicado um tarifário que pode depender de vários fatores. No caso específico da utilização de sistemas de gestão de bases de dados esses fatores poderão ser: o espaço de armazenamento utilizado, a quantidade de dados transferidos de e para o serviço e o desempenho contratado do serviço, por exemplo, em termos do número de operações por unidade de tempo.

1.2 MOTIVAÇÃO

Os serviços de armazenamento são objeto da cobrança de tarifários, que dependem de vários fatores, sendo um deles o número de operações de leitura e escrita efetuadas. No entanto, o valor cobrado pela execução destas operações pode também depender das garantias a nível de coerência com que estas são feitas. Para uma utilização ótima deste tipo de serviço, deve garantir-se que a interação com os dados respeita a coerência necessária, contudo deverá tentar-se minimizar o custo do serviço.

Por outro lado, existem muitas aplicações construídas com base numa lógica transacional que necessitam de ter acesso aos dados com garantias de coerência, pelo que torna-se útil oferecê-la também em serviços de armazenamento na nuvem. Têm por isso surgido camadas transacionais de interface com sistemas de armazenamento chave-valor que usam dois métodos principais: o **conservador**, em que transações que poderão entrar em conflito são detetadas e impedidas de executar concorrentemente e o **otimista**, no qual as transações são executadas concorrentemente, independentemente de possíveis conflitos, que apenas são detetados e solucionados antes da confirmação da transação. Estas camadas pela sua natureza medeiam todos os acessos ao sistema na nuvem mas dependem das garantias de acesso coerente aos dados na nuvem.

1.3 OBJETIVO

O objetivo deste trabalho é validar e quantificar a possibilidade de reduzir o custo de acesso a serviços de gestão de bases de dados na nuvem, tirando partido do pressuposto de todos os acessos ao serviço na nuvem serem mediados por uma camada de *middleware*, sem que isso comprometa as garantias de coerência.

O custo entende-se aqui tanto em termos económicos, relativo ao valor pago pelos acessos efetuados, bem como em termos de desempenho. O pressuposto da mediação de todos os acessos justifica-se pela existência de uma concretização de transações como camada adicional sobre o serviço base. O respeito pelas garantias de coerência exige-se para satisfazer os requisitos dessa mesma camada transacional.

1.4 CONTRIBUIÇÕES E RESULTADOS

A contribuição principal deste trabalho é a quantificação da relação entre o benefício que pode ser obtido pela mediação dos acessos a um serviço de gestão de dados na nuvem e a latência do mecanismo de coordenação necessário para garantir a manutenção da coerência. Esta relação é concretizada com medições do desempenho de um serviço de gestão de dados na nuvem.

O resultado principal deste trabalho é o projeto e concretização de um modelo de simulação que descreve e reproduz o comportamento de uma camada de *middleware* para acesso coerente a serviços de armazenamento na nuvem, tendo em conta os diferentes níveis de coerência que podem ser utilizados na execução de operações, de forma a minimizar o custo de utilização do serviço.

1.5 ESTRUTURA DO DOCUMENTO

O documento contém, no capítulo 2, o estado da arte, no qual são abordados temas como as garantias de transacionais em serviços de gestão de dados tradicionais, modelos de coerência oferecidos por serviços de armazenamento na nuvem, assim como, a existência de camadas de interface com serviços de gestão na nuvem e de que forma estas possibilitam as garantias transacionais.

Segue-se o capítulo 3 onde são mostrados resultados do desempenho de serviços na nuvem e respetivo modelo de coerência.

O capítulo 4 contém a solução proposta, assim como a respetiva arquitetura e alguns pormenores da implementação do *middleware* elaborado.

A avaliação e discussão dos respetivos resultados da solução são apresentadas no capítulo 5.

O documento é então finalizado, no capítulo 6, com as conclusões retiradas acerca do trabalho desenvolvido e com possíveis sugestões para trabalho futuro.

2

ESTADO DA ARTE

As garantias de coerência oferecidas, tipicamente, pelos serviços de gestão de dados na nuvem diferem bastante das garantias transacionais existentes nos sistemas tradicionais. Neste capítulo caraterizam-se as garantias de coerência tando destes novos serviços de gestão de dados bem como as garantias transacionais tradicionais. Finalmente, descreve-se a possibilidade de obter as garantias transacionais sobre os sistemas de gestão de dados na nuvem, através de uma camada adicional de *middleware* e em que medida essa possibilidade depende das garantias de coerência já existentes para atingir o seu objetivo.

2.1 A COERÊNCIA NA NUVEM

2.1.1 *Sistemas de armazenamento chave-valor*

As bases de dados não relacionais possibilitam que sejam guardados e manuseados grandes volumes de dados, garantindo escalabilidade, assim como, disponibilidade dos mesmos e baixa latência. Contudo, os serviços que adotam este paradigma oferecem ainda fraca coerência nas operações efetuadas sobre estes.

Os sistemas de armazenamento chave-valor suportam replicação e, embora esse fator seja positivo aquando de, por exemplo, falhas de ligação, maioritariamente, estes prescindem da coerência dos dados, em favor da disponibilidade e de tolerância à partição (*partition tolerance*).

O teorema ***Consistency, Availability, Partition tolerance* (CAP)** (Brewer, 2000)

- Coerência (*Consistency*): quando há uma leitura é garantido que o seu resultado é o da operação de escrita mais recente;
- Disponibilidade/Eficiência (*Availability*): a resposta será dada em tempo razoável e sem a ocorrência de erro;
- Tolerância à partição (*Partition tolerance*): o sistema continuará a funcionar, mesmo quando há replicação;

diz que impossível que um sistema distribuído garanta mais do que duas das três propriedades referidas e, uma vez presente a partição, é óbvio que a escolha de uma segunda propriedade é entre a disponibilidade e a coerência, sendo que, predominantemente a escolha é a disponibilidade dos dados. Uma outra desvantagem que se associa diretamente às bases de dados não relacionais é a falta de suporte a transações Atomicidade, Coerência, Isolamento, Durabilidade (ACID).

Tal como já foi referido anteriormente, as empresas que fornecem serviços de armazenamento na nuvem cobram tarifários segundo vários fatores, sendo um deles o número de operações de leitura e escrita. Além disso, as empresas mencionadas oferecem também diferentes níveis de coerência para as transações, como é o caso de operações inevitavelmente coerentes (Vogels, 2009), que poderão ser um exemplo e uma ajuda na diminuição do custo das várias operações.

Em sistemas distribuídos, existem, pelo menos, dois modelos nos quais se pode dividir a coerência, sendo eles os seguintes:

- **Coerência inevitável** (*eventual consistency*): se passado um determinado tempo não houver atualizações os nodos concordarão com os valores existentes, portanto, as leituras irão ser, inevitavelmente, do mesmo valor. Com este modelo é possível obter uma maior disponibilidade/eficiência.
- **Coerência forte** (*strong consistency*): com este nível de coerência é garantido que todos os nodos concordam com a atualização de um determinado valor, antes desse novo valor se tornar vísivel, garantindo que os valores lidos serão sempre os mais atualizados e corretos, independentemente de onde o acesso é feito.

2.1.2 *Diferenciação de categorias de coerência*

2.1.2.1 *Coerência nos dados*

Uma das soluções encontradas (Kraska et al., 2009) para o problema existente sobre o nível de coerência necessário para os dados foi a divisão destes em três diferentes categorias, sendo esse o passo inicial do funcionamento desse sistema:

- **Categoria A:** Nesta categoria inserem-se os dados que requerem que a coerência se mantenha sempre.
  - Para manter a coerência associada a esta categoria é implementado o protocolo, de controlo de concorrência conservador, *two-phase locking* (2PL), que tem um comportamento favorável quando há um grande número de conflitos.
- **Categoria B:** Os dados que constituem esta categoria são aqueles que os níveis de coerência podem variar, segundo determinados fatores, mencionados mais abaixo.

- **Categoria C:** A categoria C contém os dados nos quais a incoerência temporária é aceitável.
  - A garantia de coerência dada nesta categoria é a coerência de sessão (*session consistency*), sendo que enquanto se mantiver na sessão é dada a garantia de "ler as próprias escritas" (*read-your-own-writes*).
  - É guardado o último número de versão, sendo que se um servidor receber um valor mais antigo do que um que já leu, a situação é detetada e a leitura é efetuada novamente.
  - Ao fim de algum tempo o sistema convergirá, ou seja, é garantida a coerência inevitável.

A categoria B é o foco do trabalho em análise, uma vez que é nesta categoria que se inserem os dados que podem pertencer tanto à categoria A como à C, sendo necessária a sua adaptação, consoante o nível requerido. Para tal foram definidas várias políticas:

- Geral - *General policy*: que olha para a probabilidade de haver um conflito num determinado *data item*, atribuindo-lhe a categoria A se essa probabilidade for alta;
- Tempo - *Time policy*: os níveis de coerência são baseados no tempo. É definido um determinado valor `t` que, uma vez atingido, a categoria atribuída é alterada de coerência de sessão para coerência forte. O valor `t` é, tal como na política Geral, definido segundo a probabilidade de haver conflitos mas, neste caso, a partir de um determinado momento no tempo.

Além das duas políticas mencionadas, que podem ser aplicadas a qualquer tipo de dados, existem ainda mais três, para o caso em que os dados são numéricos, apresentadas de seguida:

- Limite fixo - *Fixed threshold policy*: quando o valor de um item é menor que um determinado limite, é dada a garantia de coerência forte;
- Demarcação - *Demarcation policy*: atribui, a cada servidor, uma determinada quantidade de um certo valor, distribuindo assim a quantidade global por vários servidores. É garantido que o valor geral nunca baixa de um determinado limite, havendo coordenação quando necessário;
- Dinâmica - *Dynamic policy*: é aplicada a coerência forte caso a probabilidade de ser violada uma determinada restrição de valor se torne alta. Essa restrição é calculada tendo em conta a frequência de atualizações no *data item* em questão e os valores que os itens contêm.

2.1.2.2 *Coerência nas operações*

Esta solução (Li et al., 2012) vem trazer a possibilidade de trabalhar com diferentes níveis de coerência, introduzindo um novo conceito denominado de ***RedBlue consistency***, que tem associado à sua definição a *RedBlue order* (que define para as operações uma ordem parcial) e também a ordem específica em que as operações poderão ser executadas localmente.

As operações executadas localmente e *lazily replicated* são consideradas operações *Blue* e são eventualmente coerentes, podendo a sua ordem ser variável. Já no que diz respeito, às operações *Red* estas são fortemente coerentes, sendo executadas em série e totalmente ordenadas.

As operações deverão ser divididas em dois grupos: as geradoras (*generator operations*) e as operações "sombra" (*shadow operations*). As primeiras determinam que transições poderão haver, e além disso geram também as *shadow operations*, que aplicam as mudanças onde for necessário.

As operações u e v são comutativas se $\forall S(estado), S + u + v = S + v + u$. Uma garantia suficiente para que seja atingida convergência de estado de um sistema *RedBlue consistent* é que seguindo uma ordem *RedBlue*, todas as operações (*Red* e *Blue*) sejam comutativas.

Uma operação "sombra" poderá ou não quebrar determinados invariantes. Não são quebrados invariantes caso a operação, iniciando-se num estado válido, quando faz transições no sistema passa o mesmo para um estado válido.

Assim a decisão da "cor" das operações é a seguinte:

- caso um par de operações "sombra" u e v sejam comutativas, ambas serão *Red*;
- caso sejam quebrados invariantes, a operação "sombra" será também *Red*;
- a todas as restantes operações, que não são *Red*, é-lhes atribuída a categoria *Blue*.

A Figura 1 mostra o mencionado acima.

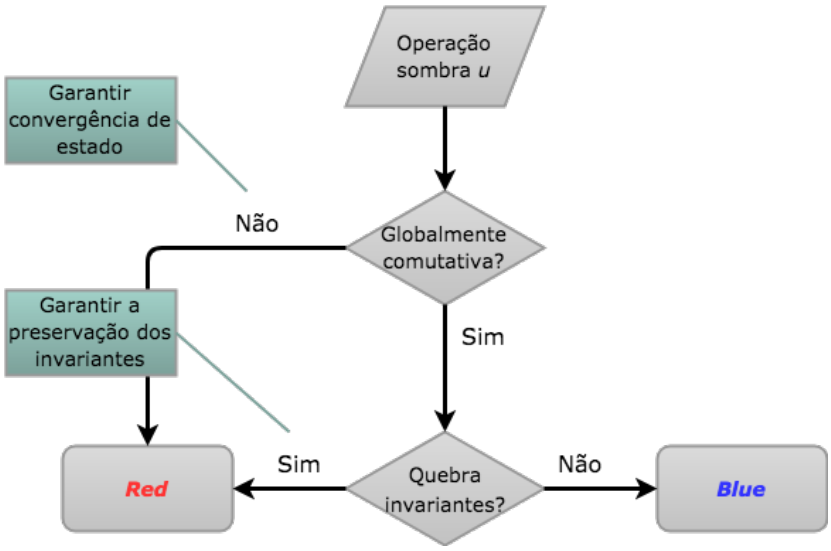


Figura 1: Decisão sobre a coerência atribuída às operações.

2.1.3 *Amazon DynamoDB*

A Amazon é uma das empresas que fornece o tipo de serviços mencionado, apresentando, entre outros, o **Amazon DynamoDB** (DynamoDB, 2012). Neste serviço, o preço das operações diferencia-se segundo o tipo de operação: leitura ou escrita.

- **Operação de leitura:**
  - A unidade de capacidade destas operações terão custo diferente dependendo se leitura é coerente ou inevitavelmente coerente, sendo que para a primeira poderá ser cobrada uma unidade de capacidade por segundo, já relativamente à segunda, poderão ser cobradas duas unidades de capacidade, ambas com um limite de itens que tenham até 4 *Kilobyte* (kB) de tamanho.
  - Como calcular:

    $$\textit{Unidades de capacidade necessárias para leitura} = \textit{Número de leituras de itens por segundo} \times \textit{tamanho do item em blocos de } 4\ kB$$

    A fórmula apresentada acima é para o caso de as leituras serem coerentes, pois para leituras inevitavalmente coerentes a taxa de transferência corresponderá ao dobro.

- **Operação de escrita:**

- Itens com até 1 kB de tamanho poderão ser executados por segundo, correspondendo assim a uma unidade de capacidade de escrita.
- Como calcular:

$$\textit{Unidades de capacidade necessárias para escrita} = \textit{Número de escritas de itens por segundo} \times \textit{tamanho do item em blocos de } 1\ \textit{kB}$$

2.1.4 *Azure Cosmos DB*

Também a Microsoft oferece um serviço que permite a escolha entre diferentes níveis de coerência, denominado de **Azure Cosmos DB** (Azure Cosmos DB, 2017).

O serviço é cobrado em termos de *Request Unit* (RU), o que significa que 1 RU de *throughput* é o mesmo que um `GET` a um documento de 1 kB.

Os cinco modelos diferentes de coerência disponibilizados no Azure Cosmos DB são então os seguintes:

- ***Strong:*** É garantido que a leitura é efetuada à última versão do item (*linearizability*).

  Para uma operação de escrita ser visível é garantido que existe concordância entre a maioria do quorum de réplicas.

- ***Bounded staleness:*** Com este modelo, a coerência de uma leitura é dada segundo um de dois fatores: um intervalo de tempo `t` ou até atingir `K` versões entre as operações de escrita. A nível de custo, uma leitura com este modelo de coerência é o mesmo que com *Strong consistency*, no que concerne ao número de RUs consumidas.

- ***Session:*** Tal como o nome indica, a coerência neste modelo é dada dentro da sessão de cada cliente. Este é o nível de coerência recomendado para quando se querem garantias como: leituras e escritas monotónicas e *Read Your Own Writes*.
  No que diz respeito ao custo das operações de leitura, este é menor do que com os modelos de coerência *Bounded staleness* e *Strong*, mas maior do que os dois que se seguem (*Consistent Prefix* e *Eventual*).

- ***Consistent Prefix:*** Neste caso, a garantia que é dada é que se não forem feitas mais escritas, todas as réplicas irão, inevitavelmente, atingir um estado em que todas convergem. Além disso, as leituras são efetuadas, exatamente, pela ordem das escritas, ou seja, se a ordem de escrita for `A-B-C`, as leituras não poderão ser feitas `C-A-B`, ou `B-A`, etc.
  Relativamente ao custo, este é menor do que utilizando o modelo de *Session consistency*.

- ***Eventual:*** A diferença para o modelo anterior (*Consistent Prefix*) é que há a possibilidade de leituras posteriores lerem valores mais antigos que os que já haviam lido. Além disso, o custo é também menor.

À medida que são relaxadas as garantias de coerência, o custo associado às operações diminui. Contudo, aumenta a disponibilidade e a escalabilidade (nas leituras) oferecidas.

2.2 SUPORTE TRANSACIONAL EM BASES DE DADOS RELACIONAIS

Os sistemas de bases de dados relacionais têm como componente essencial o suporte a transações ACID. Como é natural, poderá haver, num sistema de armazenamento, acessos concorrentes aos seus dados, sendo necessário fazer o controlo dessa mesma concorrência.

Um dos principais objetivos do controlo de concorrência é manter a integridade e coerência da base de dados. Existem vários métodos para fazer o controlo de concorrência como, por exemplo, o bloqueante (*locking*) ou o multi-versão (*multiversion concurrency control*).

A utilização do método bloqueante significa que a concorrência é tratada através da utilização de trincos (*locks*) nos dados que determinada transação irá utilizar, seja para ler ou escrever, sendo que aqui poderão diferenciar-se os trincos partilhados (*shared locks*), que são garantidos quando a transação pretende ler um determinado item, e os trincos exclusivos (*exclusive locks*), para operações de escrita. Com a utilização do método multi-versão cada item poderá ter várias cópias pois, quando os clientes acedem à BD, lêem de um *snapshot* de um determinado instante. Caso hajam escritas, até que elas estejam confirmadas (*commited*), os restantes clientes não têm acesso às alterações efetuadas. Depende do nível de isolamento, implementado na BD, qual a versão que uma transação poderá ler de determinado item. O *snapshot isolation* é o nível de isolamento mais conhecido, no qual a versão de um item que determinada transação tem acesso é aquela que tinha desde o momento que começou.

No que diz respeito aos SGBD tradicionais, o controlo de concorrência é feito recorrendo, principalmente, a dois diferentes mecanismos: o conservador e o otimista. A diferença entre ambos está que, no primeiro, as transações que poderão entrar em conflito são previamente detetadas e impedidas de executar concorrentemente, já no segundo, as transações executam concorrentemente, mas os conflitos apenas são detetados e solucionados numa fase posterior, aquando da confirmação dessas mesmas transações.

A principal vantagem da utilização de um mecanismo otimista prende-se no facto da não utilização de trincos, o que evita alguns problemas como *deadlocks*. Além disso permite um aumento na performance e um maior nível de concorrência. No entanto, quando há conflitos, estes apenas são detetados no fim da execução das operações concorrentes, levando a que tudo o que tinha sido alterado até ao momento seja descartado, gastando também recursos desnecessários. Além disso, os mecanismos optimistas podem levar a que transa-

ções longas não sejam executadas com sucesso em ambientes com elevada carga, devido à elevada probabilidade de ocorrência de conflitos. Já no caso do mecanismo conservador, a sua principal vantagem é a deteção prévia de possíveis conflitos, evitando que a execução de operações concorrentes seja levada até ao fim, desnecessariamente. Contudo, a principal desvantagem é que a execução concorrente de operações que poderiam não entrar em conflito é limitada, reduzindo assim o desempenho do sistema.

2.3 Suporte transacional em sistemas de armazenamento chave-valor

De forma a introduzir o suporte transacional em sistemas de armazenamento na nuvem, surgiram certificadores externos. Estes têm, entre outras contribuições, o objetivo de garantir que as propriedades ACID das transações sejam garantidas.

O controlo de concorrência é abordado, de várias formas, pelos certificadores.

2.3.1 *Controlo de concorrência não bloqueante (lock free)*

Nesta secção, são analisados dois certificadores externos não bloqueantes.

2.3.1.1 *Omid*

O Omid (Ferro et al., 2014) apresenta-se como sendo uma ferramenta que vem trazer suporte a transações a sistemas de armazenamento chave-valor. Este sistema utiliza um esquema centralizado e implementa *Snapshot Isolation* (SI), garantindo aos utilizadores de uma BD que, dentro de um determinado *snapshot*, terão uma visão de um estado válido da mesma. Da mesma forma, são dadas as garantias de coerência necessárias para as transações.

Tal como em outros sistemas com lógical transacional, é necessário haver controlo de concorrência, pelo que a escolha recaiu sobre o controlo de concorrência não bloqueante, ou seja, o mecanismo utilizado para o controlo de concorrência é otimista.

No caso do Omid as transações são executadas por clientes *stateless*, o que significa que o estado do cliente não tem de ser persistente[1] e em caso de falha de um cliente a segurança não fica comprometida[2].

Através de um servidor centralizado, cada transação recebe dois números de versão (*timestamps*) diferentes: um antes de ler (`ST`) e um antes de ser confirmada (`CT`).

1 Propriedade de *liveness* que declara que alguma coisa irá acontecer (Lamport, 1977). Por exemplo, um programa irá terminar se o respetivo *input* for correto.

2 Propriedade de segurança que diz que existe algo que não irá acontecer (Lamport, 1977). Isto é, se um programa começou com um *input* correto, não poderá terminar caso não tenha um resultado correto.

Conflitos entre transações ocorrem quando existe sobreposição espacial (em que duas transações `A` e `B` querem escrever na mesma linha da BD) e também sobreposição temporal (`ST(A) < CT(B)` e `ST(B) < CT(A)` : acontece quando o número de versão de confirmação de uma transação `B`, que começou depois de uma transação `A`, é menor do que a confirmação dessa mesma transação `A`.).

De forma a implementar SI são guardados alguns metadados das transações: uma lista de números de versão iniciais (*start timestamps*), uma lista de números de confirmação (*commit timestamps*) das transações e também uma lista das transações que modificaram cada linha da BD. Uma vez que em grandes bases de dados, seria impraticável guardar todas estas listas, no Omid a solução encontrada foi truncar as listas possíveis, guardando apenas a lista de *timestamps* das últimas transações aplicadas a cada linha da BD, assim como o *timestamp* máximo (`Tmax`). Além disso são também guardadas listas das transações revertidas e "em progresso" (*aborted* e *uncommitted*) que, consequentemente, levam a que se saiba quais as transações que já foram confirmadas (que são as que não pertencem nem a uma lista, nem a outra, das mencionadas). Mesmo assim, e de forma a evitar que estas listas cresçam indefinidamente, é necessário truncá-las. No caso da lista de transações não aplicadas (*uncommitted*), as transações que ultrapassem um determinado tempo, antes do `Tmax` aumentar o seu *timestamp* inicial, e desta forma a lista de transações *uncommitted* corresponde verdadeiramente às transações que se encontram em progresso. Já no que diz respeito à lista de transações revertidas, estas são excluídas da lista assim que as suas escritas são removidas da base de dados, sendo também garantido que a atualização desta lista não interfere com as transações em progresso, uma vez que esta é feita apenas quando terminam as transações que já haviam começado.

A deteção de conflitos é feita através da comparação entre o último *timestamp* de confirmação, numa determinada linha, e o *timestamp* inicial da transação em questão, verificando se o primeiro valor é menor que o segundo, sendo que nesse caso a transação é confirmada, caso contrário a transação é revertida.

Na figura 2 pode ver-se o estado inicial do sistema, na qual consta também o *Status Oracle*, onde são mantidos os metadados necessários para a deteção de conflitos entre transações. Quando é introduzida uma transação `T1`, é-lhe atribuído um número de versão inicial (*Start Timestamp* (ST)), e assim que acaba de executar é guardado o número de versão de confirmação (*Commit Timestamp* (CT)), o qual é registado como *Last Commit* (LC) em várias tabelas: a que regista os números de versão das transações, a que guarda as várias linhas do sistema de armazenamento (*rows*) com o respetivo número de versão da última transação confirmada (*last commit*) naquela linha e no próprio sistema de armazenamento. Além disso é também guardado o `Tmax`, que corresponde ao último número de versão inicial (ST) registado.

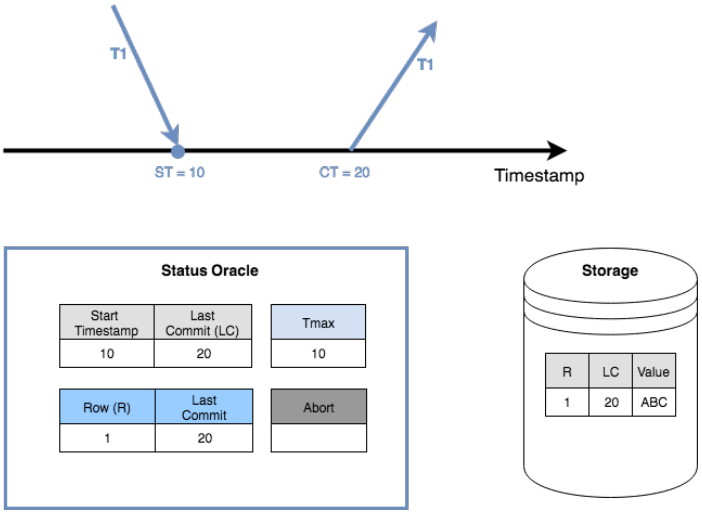


Figura 2: Estado inicial.

De seguida, uma transação T2 com ST igual 15 tenta alterar o valor da Linha 1. No entanto, a última transação confirmada naquela linha tinha o número de versão igual a 20, consequentemente, é detetado um conflito, levando a que a transação T2 seja revertida, tal como se pode verificar na ilustração 3 (em que esta fica inserida na lista de transações revertidas - "Abort"):

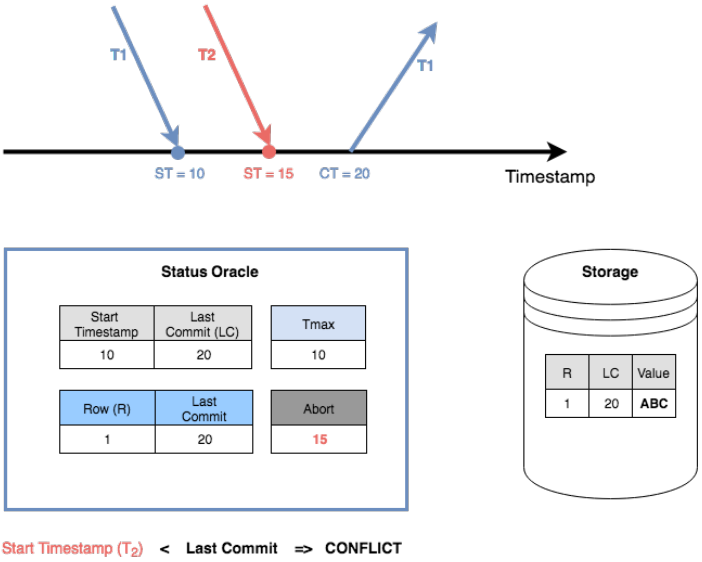


Figura 3: Deteção de conflito.

2.3.1.2 *CloudTPS*

O sistema CloudTPS (Zhou et al., 2012) apresenta-se como um gestor de transações escalável, para executar transações ACID em ambientes *cloud*. A solução proposta passa por replicar os dados e estados transacionais do gestor de transações em vários gestores de transações locais (LTM).

A implementação das transações é feita através do protocolo ***two-phase commit***, em que na primeira fase o coordenador (LTM Coordinator) solicita todos os LTMs envolvidos na transação, ficando a saber se a transação pode ser executada corretamente. Na segunda fase e, uma vez obtidas respostas favoráveis de todos os LTMs, a transação é confirmada, caso contrário é revertida.

As várias propriedades são garantidas da seguinte forma:

- **Atomicidade:** Utilizando o protocolo *two-phase commit*, tal como explicado anteriormente;
- **Coerência:** Se uma transação é executada sobre uma BD, internamente coerente, deverá deixá-la num estado coerente quando terminar a sua execução. Esta propriedade é então garantida se todas as transações executarem corretamente;
- **Isolamento:** Se duas transações têm conflitos, através da sua decomposição em pequenas sub-transações, estas são executadas sequencialmente, sendo que isto é feito utilizando *timestamp ordering*, ou seja, uma sub-transação apenas poderá executar quando todas as outras sub-transações com *timestamps* menores forem confirmadas;
- **Durabilidade:** No caso do CloudTPS esta propriedade é satisfeita uma vez que todas as alterações feitas por transações já confirmadas são escritas.

Tal como referido, cada LTM guarda uma lista das suas sub-transações e o respetivo *timestamp* atribuído, sendo este usado para fazer controlo de concorrência, recorrendo ao protocolo *timestamp ordering*. Com isto, um estado inicial possível é o apresentado na Figura 4.

De seguida, é introduzida na `LTM1` uma sub-transação da transação `T3` que tem um *timestamp* menor do que as que lá constavam (Figura 5). Posto isto, e caso as sub-transações com *timestamp* maior já tenham sido confirmadas, o conflito é comunicado ao *Local Transaction Manager Coordinator*, de forma a poder ser resolvido.

Assim sendo, a transação `T3` deverá ser revertida e obter um novo *timestamp*, tendo as respetivas sub-transações de recomeçar as suas execuções, tal como se pode observar na Figura 6.

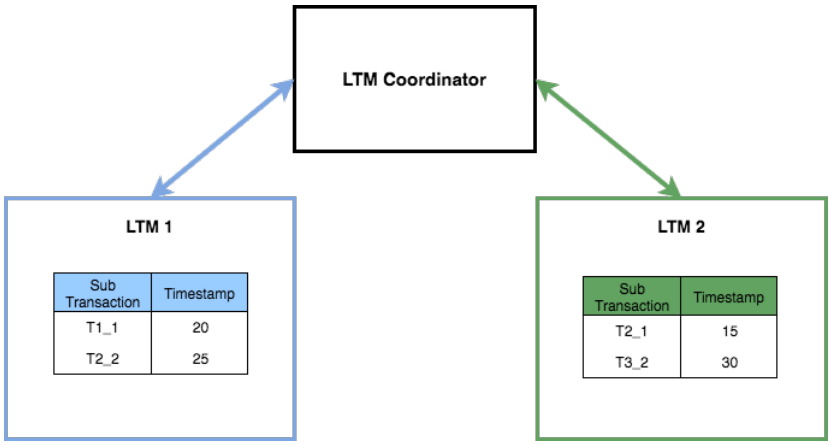


Figura 4: Estado inicial possível do sistema CloudTPS, com dois LTMs.

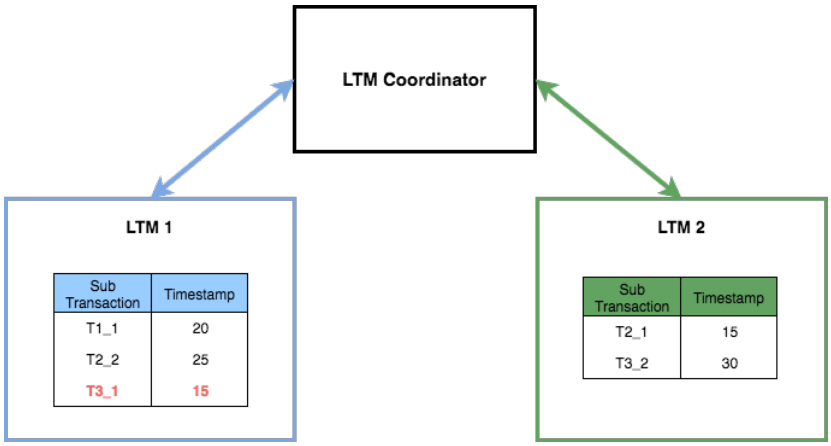


Figura 5: Deteção de conflito.

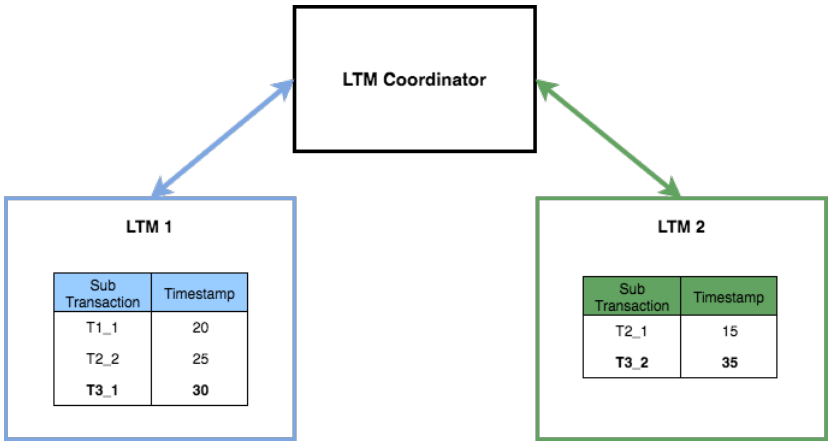


Figura 6: Estado final da execução.

Tal como já foi referido, a maior parte dos sistemas disponíveis na nuvem optam por oferecer disponibilidade dos dados ao invés da sua coerência, no entanto, neste certificador a escolha foi a do favorecimento da coerência.

As leituras podem ser fortemente coerentes, contudo o sistema apresenta uma solução em que essas operações poderão ler de um *snapshot* coerente mais antigo, introduzindo o conceito de *Weakly-Consistent Read-only Transaction* (WCRT), que permite qualquer tipo de consulta (incluindo *range queries*[3]). Estas transações são implementadas através de um mecanismo que atribui ao *snapshot* um identificador único, sendo que o Transaction Processing System (TPS) comunica o Identificador único (ID) do último *snapshot* correto, para que as operações sejam coerentes.

Quanto às operações de escrita, se o sistema de armazenamento em utilização assegurar o *Read-Your-Writes*, essa coerência é suficiente, caso contrário, a garantia da coerência fica a cargo dos LTMs.

2.3.2 *Controlo de concorrência bloqueante (com locks)*

2.3.2.1 *Deuteronomy*

A abordagem do sistema Deuteronomy (Levandoski et al., 2011) para o suporte de transações ACID na nuvem é feita através de dois componentes diferentes: o componente transacional (*Transactional Component* (TC)) que gere as transações e o respetivo controlo de concorrência entre elas; e o componente de dados (*Data Component* (DC)), onde se encontram guardados os dados, assim como as mudanças efetuadas nos itens. Mesmo que

3 Uma *range query* é uma operação efetuada a uma BD que tem como resultado todas as entradas que tenham um valor que se encontra dentro de um determinado intervalo, tendo em conta um dado limite inferior e um limite superior

existam vários DCs distribuídos por vários sítios, apenas existe um TC, que trata as transações.

Tanto o TC como o DC são constituídos por várias partes. No caso do TC, este divide-se em cinco elementos: *session manager*, *record manager*, *table manager*, *lock manager*, *log manager*.

Os pedidos são enviados diretamente para o TC e tratados pelo *session manager*, seguidamente há uma divisão, pois no caso do pedido ser de leitura ou escrita, este é reencaminhado para o *record manager*, já se o pedido for para criar ou apagar algum registo/tabela, irá para o *table manager*. Em ambos os casos, e de forma a fazer o controlo de concorrência necessário, são também chamados o *log* e *lock managers*, sendo que é neste último que é efetuado o tratamento dos *locks* necessários, aquando da existência de conflitos.

Tal como referido acima, é o *table manager* que trata das operações de criação, modificação, etc., dos componentes das tabelas, portanto, é aqui que deverá ser garantida a coerência necessária. Cada *table manager* tem um *table catalog* e um *column catalog*, onde são guardados diversos metadados necessários. Assim, para garantir que há coerência entre as tabelas e os respetivos metadados, cada transação tem duas operações simétricas que manipulam o *table catalog* e o respetivo *column catalog*.

3

DESEMPENHO DE SERVIÇOS NA NUVEM

A avaliação da viabilidade de introdução de coordenação adicional, para reduzir o custo do acesso aos serviços de dados da nuvem, depende do desempenho e custo típicos destes serviços. Neste capítulo apresentam-se resultados de avaliação do Amazon DynamoDB, alguns já existentes na literatura e outros obtidos no âmbito deste trabalho.

Em qualquer dos casos, foi utilizado o YCSB (Cooper et al., 2010), que é uma *framework* para avaliação de diferentes serviços na nuvem, focando-se nos serviços que oferecem acesso a dados *online*, através de leituras e escritas. Os valores apresentados abaixo serão usados para calibrar a avaliação do *middleware* desenvolvido.

3.1 RESULTADOS PRÉ-EXISTENTES

Em (Klems et al., 2012) foi feita uma avaliação do Amazon DynamoDB, através da utilização do YCSB, relativamente à sua performance, sendo que o principal objetivo do trabalho em questão é a construção de uma ferramenta para a medição e análise do tempo de execução para serviços de bases de dados na nuvem. Foram abrangidas as diferentes operações: leituras coerentes, leituras eventualmente coerentes e escritas. Nesta experiência, os autores do artigo alteraram a propriedade de comprimento de registo (*record field length property*) de 100 `bytes` para 90 `bytes`, uma vez que, no DynamoDB o denominado *provisioned throughput*[1] calcula a capacidade, baseando-se em itens cujo tamanho é menor que 1 kB. Os resultados obtidos podem ser observados na tabela 1, onde se podem consultar os valores da latência média atingida.

	Throughput	Escritas - Média	Leituras - Média
C	96 ops/s	44 ms	44ms
EC	99 ops/s	16ms	15ms

Tabela 1: Resultados obtidos, com a utilização do YCSB.

1 Valor máximo de capacidade que uma aplicação pode consumir de uma tabela ou índice. Uma vez excedido esse valor os pedidos podem ser limitados (*throttled*) (AWS Documentation, 2012-08-10).

A conclusão tirada é que a latência média aumenta, aproximadamente, 200% quando são efetuadas leituras coerentes. Além disso, o *throughput* não foi além dos três dígitos, no entanto, no momento em que foram realizadas as experiências o DynamoDB ainda estaria em fase beta, tal como explicado pelos autores.

3.2 RESULTADOS OBTIDOS

Com o objetivo de obter resultados mais concretos acerca do desempenho do DynamoDB, foram elaboradas algumas experiências, variando diversos fatores. No entanto, estas foram limitadas pela parcela grátis oferecida pela Amazon, na qual apenas se podem utilizar 25 unidades de capacidade de gravação (25 escritas por segundo) e 25 unidades de capacidade de leitura (25 leituras fortemente coerentes e 50 leituras eventualmente coerentes).

O processo foi iniciado com a criação de uma tabela `usertable`, com a chave primária `FirstName`, no DynamoDB, na qual foram inseridos dados, através do comando `load` do YCSB.

Foram aprovisionadas dez unidades de capacidade, tanto para leitura como para escrita. Além disto, foram também construídas diferentes *workloads* para serem utilizadas nas experiências, tendo por base algumas das já disponibilizadas pelo YCSB.

O comando utilizado para correr as experiências foi o seguinte:

```
./bin/ycsb run dynamodb -target 1 -P workloads/workloadi -P
    ./dynamodb/conf/dynamodb.properties -p measurementtype=timeseries -p
    timeseries.granularity=1000 > results/results.dat
```

- `-target`: número de operações que irão ser feitas por segundo;
- `-P`: utilizado para fazer *load* aos ficheiros de propriedades (por exemplo, o `dynamodb.properties`);
- `workloadi`: *workload* utilizada;
- `measurementtype`: foi utilizado o `timeseries` com uma granularidade de 1000, o que significa que a cada 1000 milissegundos é comunicada a latência média atingida, dentro daquele intervalo de tempo;
- `results.dat`: ficheiro no qual irão constar os resultados.

Um exemplo de uma *workload* utilizada é a apresentada em 3.1.

```
    maxexecutiontime=3600
    fieldlength=90
```

```
workload=com.yahoo.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=1
updateproportion=0
scanproportion=0
insertproportion=0

requestdistribution=uniform
```

Excerto 3.1: *Workload* para fazer *benchmarking*, utilizando o YCSB.

Com isto consegue perceber-se de que forma poderão ser variados os parâmetros. As experiências realizadas tiveram como `maxexecutiontime` 3600 segundos (1 hora), isto é, todas elas tiveram como tempo de execução esse tempo. Também o tipo de distribuição se manteve igual, tendo sido utilizada uma distribuição uniforme, pois esta é a distribuição recomendada para o DynamoDB (YCSB - DynamoDB Binding, 2015), para que sejam atingidas a escalabilidade e performance desejadas.

Tal como no artigo apresentado na subsecção 3.1, o parâmetro `field length` (tamanho) foi reduzido para 90, uma vez que, com o valor *default* cada operação iria utilizar 2 unidades de capacidade (YCSB - DynamoDB Binding, 2015).

As proporções entre as operações de leitura (`readproportion`) e de escrita (`updateproportion`) foram os parâmetros que sofreram variações.

As letras "C" e "EC" representam a execução ou não, respetivamente, de operações de leitura coerentes.

Uma vez que se está a definir o número de operações por segundo, no comando `run` através do parâmetro `target`, o *throughput* é 1 operação por segundo.

	Leituras		Escritas	
	Nº operações (uni)	Latência média (ms)	Nº operações (uni)	Latência média (ms)
100% EC	3598	166.6208	-	-
100% C	3599	187.1147	-	-
100% Escritas (W)	-	-	3600	199.0455

Tabela 2: Resultados do YCSB para o Amazon DynamoDB: apenas leituras e apenas escritas.

	Leituras		
	Nº operações (uni)	Latência média (ms)	Desvio Padrão
10% W - 90% EC	3240	125.9405	39.1264
10% W - 90% C	3238	141.4449	78.2321

Tabela 3: Resultados do YCSB para o Amazon DynamoDB: leituras e escritas, simultaneamente - Parte 1.

	Escritas		
	Nº operações (uni)	Latência média (ms)	Desvio Padrão
10% W - 90% EC	359	125.8087	32.6969
10% W - 90% C	359	147.0031	84.8725

Tabela 4: Resultados do YCSB para o Amazon DynamoDB: leituras e escritas, simultaneamente - Parte 2.

Os resultados presentes nas tabelas 2, 3 e 4 indicam que com leituras eventualmente coerentes a latência média é menor do que com leituras coerentes, apesar da diferença na latência média se diluir devido à comunicação e de uma variabilidade grande ilustrada pelo valor do desvio padrão. Verifica-se que com a execução de leituras coerentes a latência média sofre um aumento de, aproximadamente, 12%, relativamente aos resultados obtidos com leituras eventualmente coerentes, independentemente da execução ou não de operações de escrita.

Os resultados apresentados, tanto na subsecção 3.1 como na subsecção 3.2, demonstram que existem diferenças, na latência média atingida, aquando da execução ou não de operações de leitura coerentes.

Tal como explicado no início, os valores presentes tanto na subsecção 3.1 como na subsecção 3.2 serão utilizados para a avaliação do trabalho desenvolvido. Assim, os valores apresentados em 3.1 formam a base do cenário em que a aplicação é executada na mesma infraestrutura do serviço de armazenamento. Já os resultados obtidos e demonstrados na subsecção 3.2 formam o cenário em que a aplicação executa fora da infraestrutura do serviço de armazenamento, acedendo a este remotamente.

4

SIMULADOR DE ACESSOS À NUVEM

Existem vários serviços de armazenamento na nuvem que oferecem tarifários com custos diferenciados de acordo com o nível de coerência das operações, sem suporte transacional. No entanto, surgiram camadas transacionais de interface com estes serviços que garantem as propriedades ACID às transações, que requerem coerência forte no acesso aos dados, como, por exemplo, o Omid (Ferro et al., 2014) ou o Deuteronomy (Levandoski et al., 2011) e que têm necessariamente que mediar todos os acessos aos dados. Neste capítulo propõe-se uma arquitetura que explora esta possibilidade para reduzir o custo do acesso e propõe um modelo de simulação para a avaliar.

4.1 SOLUÇÃO PROPOSTA

Depois de conhecidos modelos de coerência oferecidos por serviços de armazenamento na nuvem, o objectivo é minimizar a necessidade de fazer leituras coerentes ao utilizar este tipo de serviço, mas garantindo que é utilizada coerência forte, sempre que necessário.

A solução proposta passa pela elaboração de um *middleware* para que as leituras e escritas sejam coerentes, procurando a redução de custos, tendo em conta o custo associado à utilização do serviço.

O *middleware* proposto faz uso de uma *cache* local, assim como, é utilizada uma forma de os demais clientes saberem quando determinada chave foi escrita.

A avaliação do mesmo foi realizada com base num dos sistemas de armazenamento na nuvem que oferece custos diferenciados pelo nível de coerência das operações, tendo sido escolhido para o efeito o DynamoDB.

Como mostra a Figura 7, o sistema é constituído por vários componentes, que desempenham funções distintas.

- ***Cloud:*** contém a lista de chaves disponíveis para acesso e representa o serviço de armazenamento;
- **Clientes:** tal como num modelo cliente-servidor *standard*, no qual a função do cliente é o envio de pedidos a um determinado servidor, que é o fornecedor da informação/

serviço (Sinha, 1992). Neste caso específico, os clientes têm um conjunto de operações de leitura e escrita para efetuar, fazendo esses pedidos à *Cloud*;

  - ***Cache:*** cada cliente tem a sua própria *cache*, de forma a guardar pares chave-valor para que, posteriormente, possam ser acedidos mais facilmente, sendo que isto permite evitar pedidos ao serviço de armazenamento, reduzindo custos e latência.

- ***Middleware:*** aqui é mantida a informação sobre se o valor associado a uma determinada chave foi alterado. Este é um módulo fundamental do sistema, uma vez que todas as operações efetuadas pelos clientes têm de o consultar, aquando da respetiva execução.

4.2 IMPLEMENTAÇÃO

O modelo de simulação usado para avaliar a validade da arquitetura proposta é uma modelo de simulação por eventos que, em pormenor, funciona da seguinte forma. Sempre que há uma escrita, a chave sobre a qual esta é efetuada é marcada como alterada, o que significa que a leitura seguinte a essa mesma chave tem de ser coerente, como representado na Figura 7. Dessa forma, os clientes sabem também que o valor anterior, que poderiam ter em *cache*, não está atualizado.

A contínua alteração de chaves pode levar a que se atinja um estado em que todas as chaves estejam marcadas como alteradas. Consequentemente, isto poderá traduzir-se num aumento de custos desnecessário para os clientes, uma vez que, estes terão de efetuar todas as leituras coerentemente, a partir desse momento. Assim, foi também necessário escolher um critério para a limpeza das mesmas. Posto isto, a solução encontrada foi a limpeza desta camada segundo um determinado intervalo de tempo. Este intervalo é escolhido de forma a ser um limite superior ao tempo que demora até que as leituras eventualmente coerentes devolvam o valor mais recente.

O sistema implementado permite a variação do tamanho da *cache*, pelo que, dependendo desse fator o algoritmo utilizado para a execução de operações de leitura poderá ter comportamentos distintos. O procedimento efetuado afeta diretamente o custo de cada operação, que depende também do seu tamanho. O algoritmo 1 demonstra o procedimento efetuado para uma operação de leitura, quando a *cache* tem tamanho maior do que zero.

No caso das operações de escrita, a execução das mesmas segue um procedimento mais direto, tal como é possível observar no algoritmo 2.

Caso o tamanho da *cache* seja igual a zero, a execução de um evento de escrita segue um comportamento semelhante ao indicado em 2, ao contrário das operações de leitura que procedem de um modo diferente, tal como se pode constatar em 3.

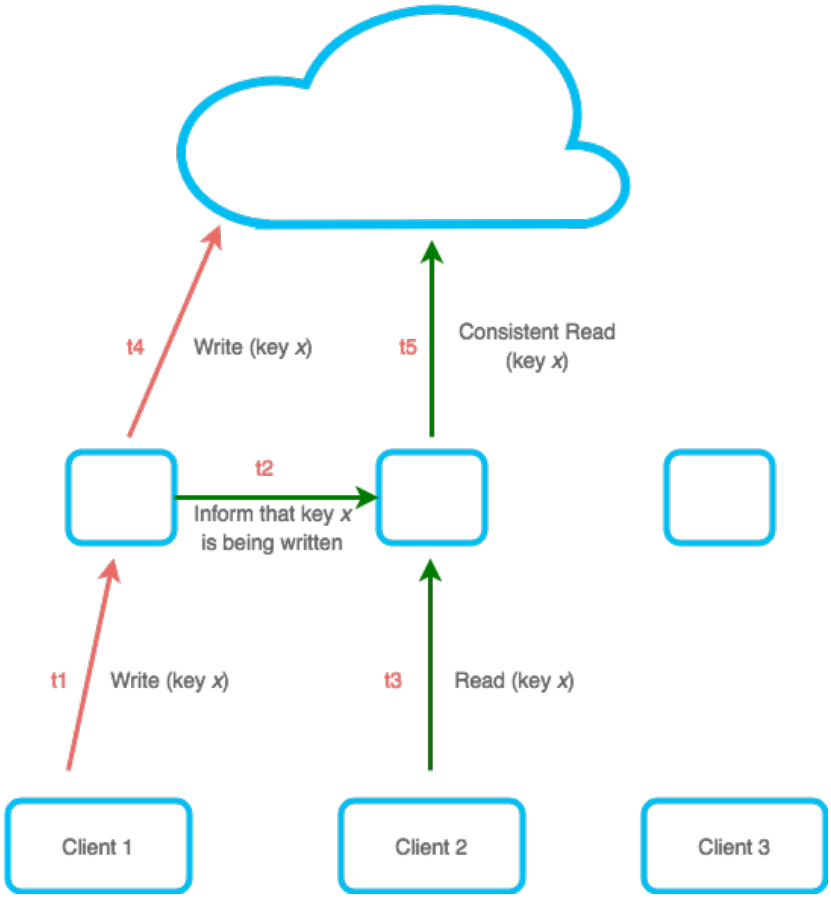


Figura 7: Marcação de uma chave como alterada, após uma escrita.

Algoritmo 1 Algoritmo para a execução de uma operação de leitura.

```
1: if key == unchanged then
2:     if cache(key) then
3:         reads_from_cache(key)
4:         cost ⇐ 0
5:     else
6:         eventual_consistent_read_from_cloud(key)
7:         cost ⇐ x
8:     end if
9: else if key == changed then
10:    consistent_read_from_cloud(key)
11:    cost ⇐ y
12: end if
```

Algoritmo 2 Algoritmo para a execução de uma operação de escrita.

```
1: write(key)
2: cost ⇐ z
3: mark_as_changed(key)
```

Algoritmo 3 Algoritmo para a execução de uma operação de leitura, sem *cache*.

```
1: if key == unchanged then
2:     eventual_consistent_read_from_cloud(key)
3:     cost ⇐ x
4: else if key == changed then
5:     consistent_read_from_cloud(key)
6:     cost ⇐ y
7: end if
```

Aquando da execução das várias operações, tal como descrito nos excertos acima (1, 2), é também efetuada a adição de valores na *cache*, caso o tamanho desta o permita, como o exemplo apresentado em 4.

Algoritmo 4 Execução de uma operação de leitura, e respetiva adição do valor na *cache*.

```
1: if key == unchanged then
2:     if cache(key) then
3:         ...
4:     else
5:         eventual_consistent_read_from_cloud(key)
6:         cost ⇐ x
7:         if not_full(cache) then
8:             cache_add(key)
9:         end if
10:    end if
11: end if
```

Como é natural, há a possibilidade que, a determinado momento, a utilização máxima da *cache* seja atingida, no entanto, é fundamental que novos valores possam ser adicionados. Assim sendo, a solução proposta passa pelo seguinte: cada vez que é efetuada uma leitura a um determinado valor, presente em *cache*, é atualizado o *timestamp* respetivo. Então, quando a *cache* se encontra cheia e surge uma tentativa de inserção de um novo valor são escolhidas aleatoriamente duas chaves e é removida aquela com o *timestamp* mais antigo, para dar lugar a uma nova chave (Azar et al., 1994). Esta sequência de acontecimentos está demonstrada nos algoritmos 5 e 6.

Algoritmo 5 Execução de uma operação de leitura à *cache*, e respetiva atualização do timestamp do valor presente em *cache*.

```
1: if key == unchanged then
2:     if cache(key) then
3:         reads_from_cache(key)
4:         key_timestamp ⇐ now()
5:     end if
6: end if
```

Algoritmo 6 Execução de uma operação de escrita, e respetiva adição do valor na *cache*, depois de removido um valor antigo.

```
1: write(key)
2: ...
3: if not_full(cache) then
4:     cache_add(key)
5: else
6:     unveil(key_1)
7:     unveil(key_2)
8:     if key_1_timestamp before key_2_timestamp then
9:         remove_from_cache(key_1)
10:    else
11:        remove_from_cache(key_2)
12:    end if
13:    cache_add(key)
14: end if
```

Além de fatores mencionados como, por exemplo, a existência ou não de determinado valor na *cache*, ou até o tamanho da mesma, um outro coeficiente no cálculo do custo de uma operação é o seu tamanho. Tal como acontece no DynamoDB, onde operações com um tamanho superior a 4 kB terão o dobro do custo, também no sistema desenvolvido esse foi um fator tido em conta. Assim sendo, pode verificar-se no algoritmo 7 um exemplo de como será calculado o custo de uma operação que ultrapasse um determinado tamanho, tido como máximo para que não tenha custo a dobrar.

Algoritmo 7 Algoritmo para a execução de uma operação de leitura, com custo dobrado.

```
1: if key == unchanged then
2:     if cache(key) then
3:         reads_from_cache(key)
4:         cost ⇐ 0
5:     else
6:         eventual_consistent_read_from_cloud(key)
7:         if event_size > max_size then
8:             cost ⇐ 2x
9:         else
10:            cost ⇐ x
11:        end if
12:    end if
13: else if key == changed then
14:    consistent_read_from_cloud(key)
15:    if event_size > max_size then
16:        cost ⇐ 2y
17:    else
18:        cost ⇐ y
19:    end if
20: end if
```

5

RESULTADOS DA SIMULAÇÃO

Este capítulo descreve a avaliação feita usando o modelo de simulação, com os respetivos resultados. As simulações foram elaboradas, com o objetivo de compreender, com os resultados obtidos, de que forma o trabalho desenvolvido potencia a redução de custos, aquando da utilização de serviços de armazenamento na nuvem, tendo em conta os tarifários oferecidos, que possibilitam diferentes níveis de coerência na execução de operações de leitura e escrita. Para tal utilizaram-se os algoritmos descritos na secção 4.2, assim como a arquitetura presente na secção 4.1.

5.1 PARÂMETROS

As simulações foram feitas recorrendo à variação dos parâmetros mencionados abaixo:

- número de *runs*;
- número de clientes;
- número de chaves;
- número de eventos;
- percentagem de operações de leitura;
- percentagem de operações de escrita;
- tamanho de *cache*;
- intervalo entre limpezas;
- tempo para efetuar uma escrita;
- tempo para efetuar uma leitura coerente;
- tempo para efetuar uma leitura eventualmente coerente;
- tempo para marcar um determinado item como modificado.

Sabe-se que o custo de uma determinada operação depende de dois fatores: **tamanho** e **tipo**.

Tal como referido na secção 4.2, a execução das operações acarreta determinados custos, que foram definidos da seguinte forma:

- Leitura à *cache*: 0 Unidades Monetárias (UM);
- Leitura eventualmente coerente: 1 UM;
- Leitura coerente: 2 UM;
- Escrita: 2 UM.

Além disso, foi mencionado na mesma secção que caso a operação atingisse um determinado tamanho, o seu custo seria dobrado, consequentemente, uma leitura eventualmente coerente passa a custar 2 UM, uma leitura coerente 4 UM e uma escrita 4 UM.

Para o tamanho das operações, o valor pode variar entre 1 e 10, sendo o valor aleatoriamente atribuído. Uma vez atingindo um valor acima de 5, isto, é entre 6 e 10, a operação tem então o custo a dobrar.

Posto isto, é possível apresentar os resultados obtidos, onde se modificam diferentes variáveis, sendo elas: o tamanho da *cache*, a percentagem entre o número de leituras e escritas, o intervalo entre limpezas, o tempo que cada operação demora, o tempo de marcar um item como modificado.

5.2 Acesso remoto à nuvem

5.2.1 *Utilização da cache*

De seguida apresentam-se os resultados relativos à utilização da *cache*, tendo em conta a variação do seu tamanho. Poderão observar-se gráficos correspondentes ao tempo médio por evento, ao custo médio por evento, assim como a percentagem de ocorrências das diferentes operações (escritas, leituras coerentes, leituras eventualmente coerentes, leituras feitas à *cache*).

Os valores utilizados para o tempo para efetuar as diversas operações foram escolhidos tendo em consideração os resultados obtidos na secção 3.2.

Assim sendo a *workload* utilizada foi a seguinte:

```
Número de runs: 1;
Número de clientes: 10;
Número de chaves: 1000;
Número de eventos: 10000;
```

```
Intervalo entre limpezas: 100 ms;
Tempo para efetuar uma escrita: 147 ms;
Tempo para efetuar uma leitura coerente: 141 ms;
Tempo para efetuar uma leitura eventualmente coerente: 126 ms;
Tempo para marcar um determinado item como modificado: 0 ms.
```

Excerto 5.1: *Workload* para simular o impacto da *cache*, no acesso remoto à nuvem.

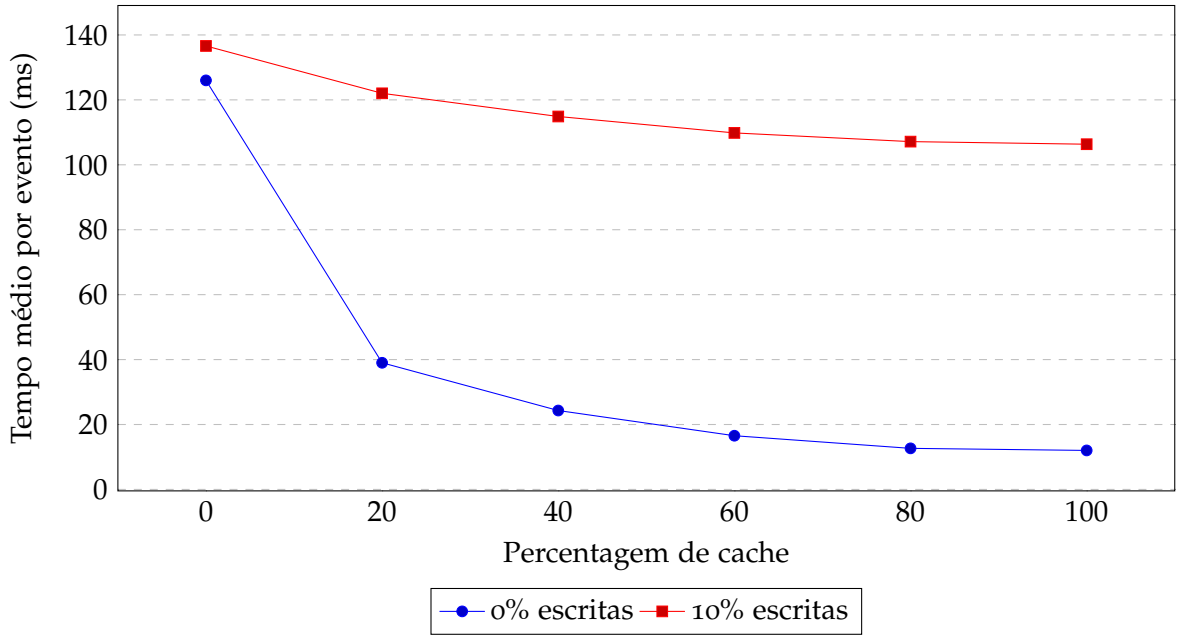


Figura 8: Tempo médio por evento, com a variação do tamanho da *cache*.

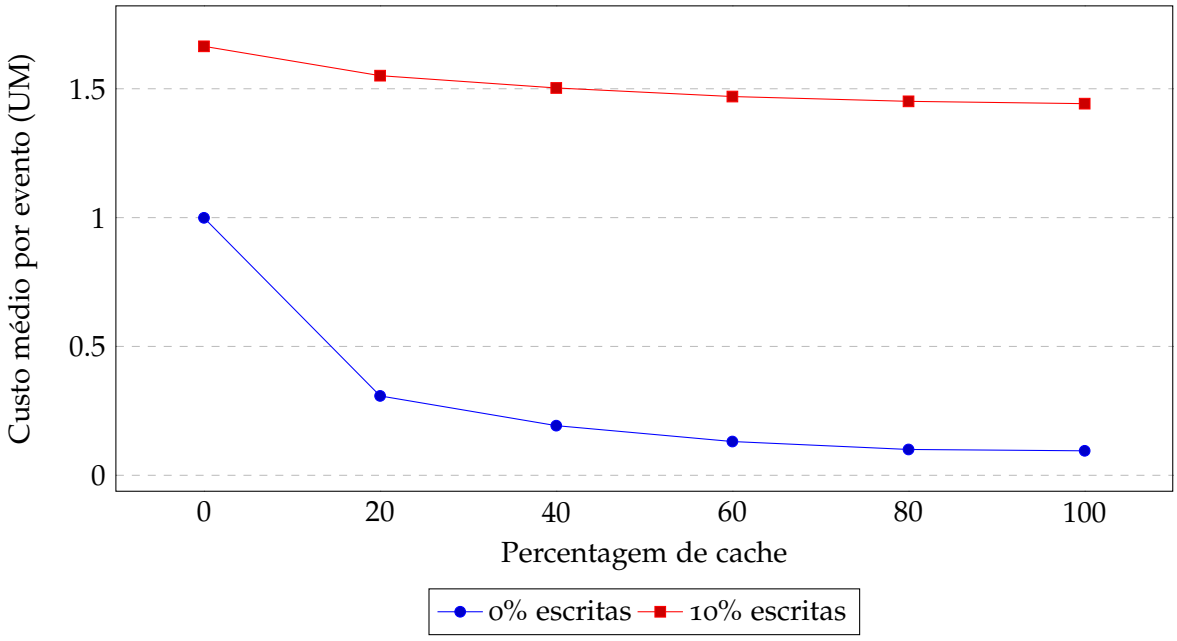


Figura 9: Custo médio por evento, com a variação do tamanho da *cache*.

Os gráficos 8 e 9 mostram que com o aumento do tamanho da *cache* o tempo médio e o custo médio, respetivamente, por evento diminuem.

Quando são efetuadas apenas leituras, a descida tanto em termos de tempo médio como de custo médio é de, aproximadamente, 90.5%. Já com 10% de escritas e 90% de leituras, entre o valor zero de tamanho da *cache* e o máximo, a descida em termos de tempo médio é de 22% e o custo médio por evento tem uma descida de 13%.

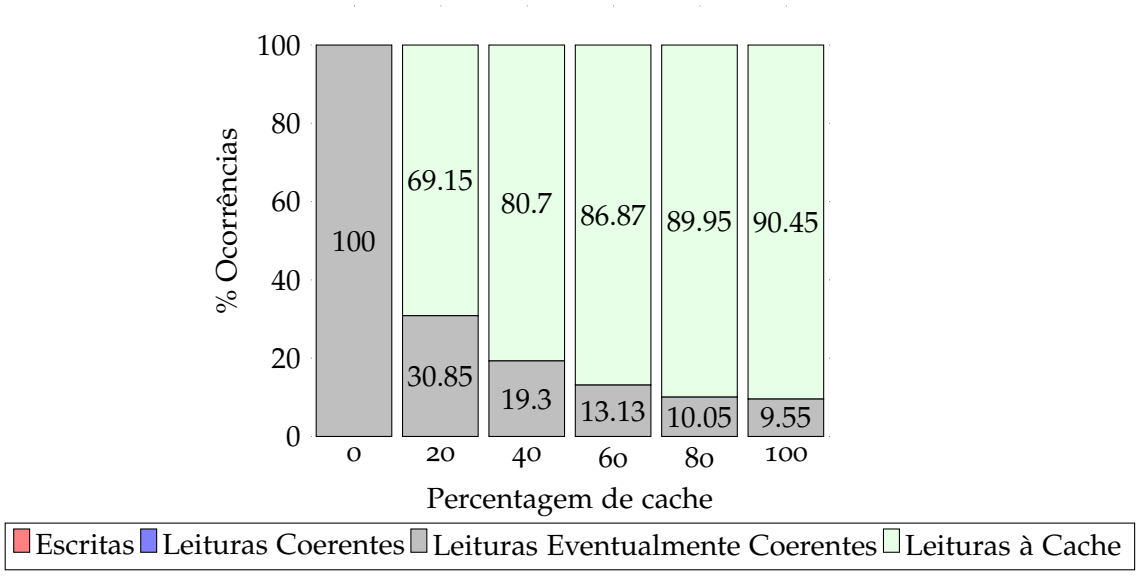


Figura 10: Percentagem de ocorrências de cada operação, com a variação do tamanho da *cache*, com 100% de leituras.

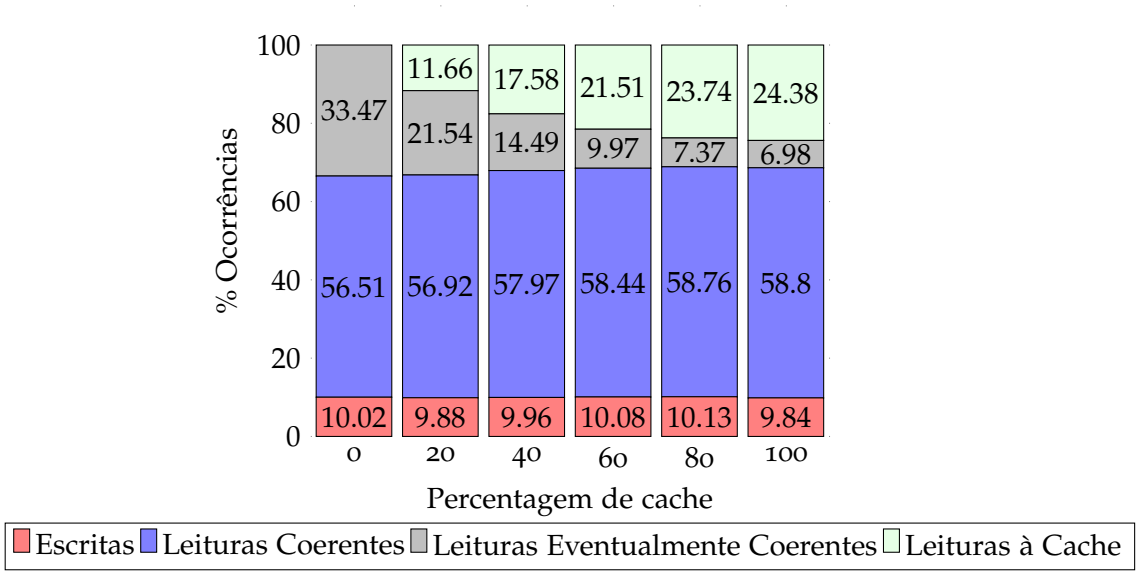


Figura 11: Percentagem de ocorrências de cada operação, com a variação do tamanho da *cache*, com 90% de leituras e 10% de escritas.

Como é expectável, o aumento da percentagem de leituras satisfeitas pela *cache* acompanha o aumento do tamanho da *cache*.

5.2.2 *Tempo para a marcação de um item como modificado*

A *workload* utilizada foi a seguinte:

```
Número de runs: 1;
Número de clientes: 10;
Número de chaves: 1000;
Número de eventos: 10000;
Percentagem de operações de leitura: 90%;
Percentagem de operações de escrita: 10%;
Intervalo entre limpezas: 100 ms;
Tempo para efetuar uma escrita: 147 ms;
Tempo para efetuar uma leitura coerente: 141 ms;
Tempo para efetuar uma leitura eventualmente coerente: 126 ms.
```

Excerto 5.2: *Workload* para a simulação da marcação de um item como modificado.

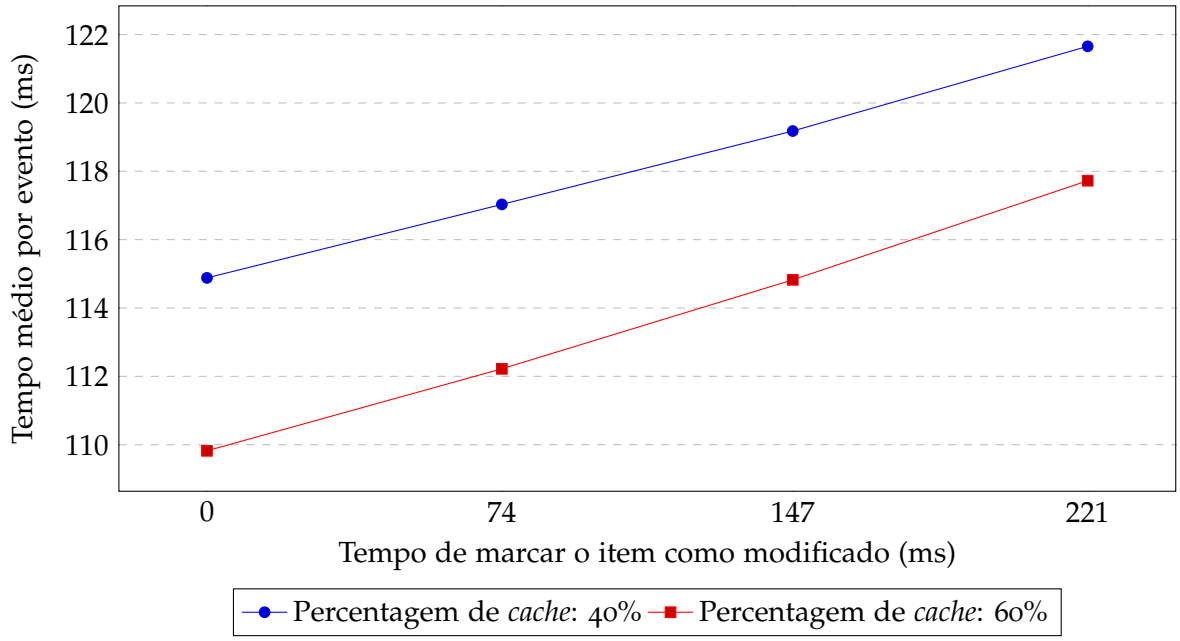


Figura 12: Variação do tempo médio por evento, com a variação do tempo de marcação de um item como modificado.

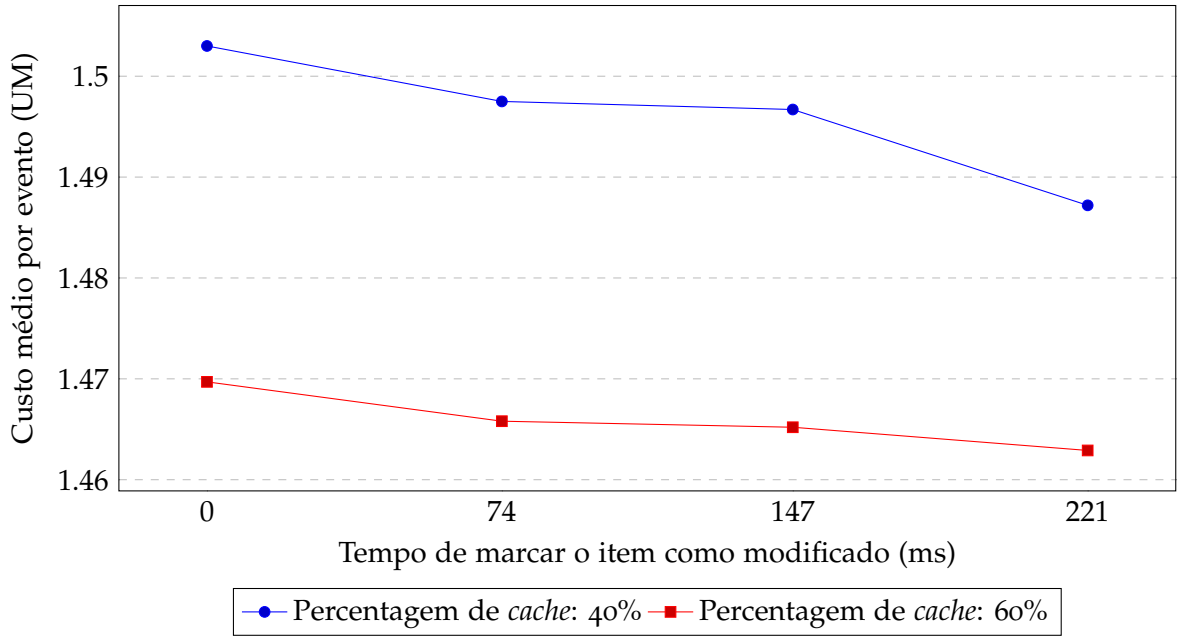


Figura 13: Variação do custo médio por evento, com a variação do tempo de marcação de um item como modificado.

Foram feitas simulações com dois valores diferentes de tamanho de *cache*, sendo possível constatar, através dos gráficos 12 e 13, que o aumento do tempo para a marcação de uma chave como alterada, após uma escrita, tem impacto na performance em questões de

tempo gasto na execução das operações. Contudo, esse mesmo aumento traduz-se numa diminuição do custo.

É, portanto, possível concluir que abdicando de algum desempenho em termos de tempo de execução, se consegue poupar em termos monetários.

5.3 ACESSO LOCAL À NUVEM

5.3.1 *Utilização da cache*

Os valores utilizados para o tempo para efetuar as diversas operações foram escolhidos tendo em consideração os resultados obtidos na secção 3.1.

Assim sendo a *workload* utilizada foi a seguinte:

```
Número de runs: 1;
Número de clientes: 10;
Número de chaves: 1000;
Número de eventos: 10000;
Intervalo entre limpezas: 30 ms;
Tempo para efetuar uma escrita: 44 ms;
Tempo para efetuar uma leitura coerente: 44 ms;
Tempo para efetuar uma leitura eventualmente coerente: 16 ms;
Tempo para marcar um determinado item como modificado: 0 ms.
```

Excerto 5.3: *Workload* para simular o impacto da *cache*, no acesso local à nuvem.

Utilizando os valores mencionados na *workload* 5.3 conseguem obter-se resultados que demonstram uma descida de 10% no tempo médio de execução de um evento (gráfico 14) e de 14% no custo médio por evento (figura 15), quando a proporção de leituras e escritas é 90% - 10%.

Quando apenas são efetuadas leituras, os resultados são semelhantes aos mencionados na secção 5.2.1, ou seja, uma descida de 90.5%, tanto em tempo médio como em custo médio.

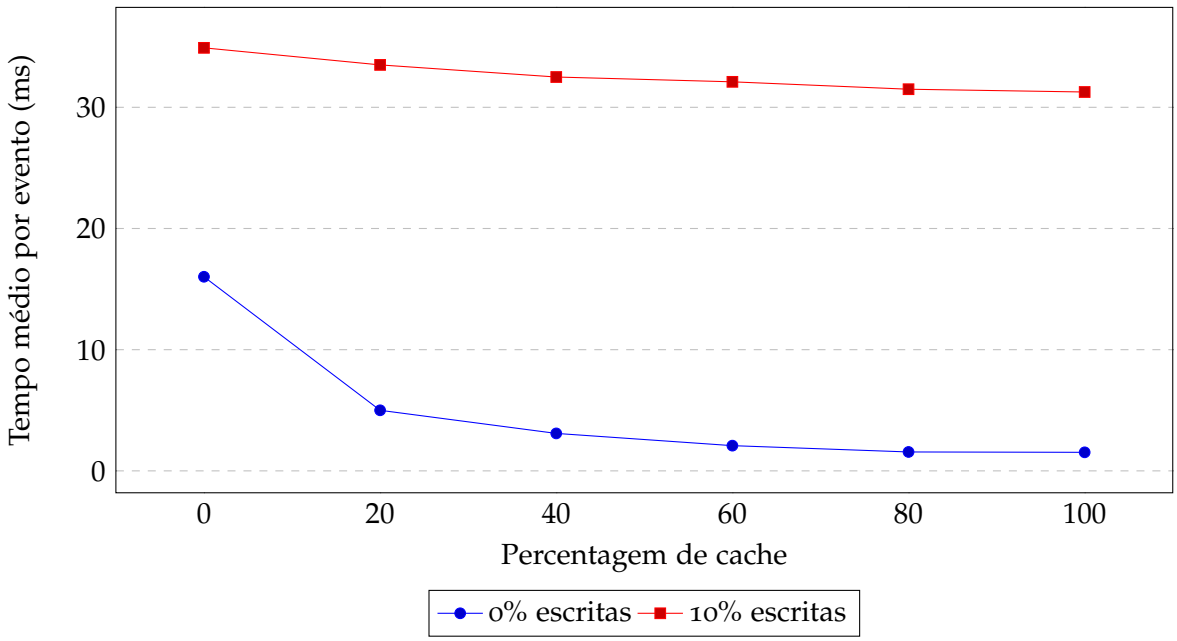


Figura 14: Variação do tempo médio por evento, com a variação do tamanho da *cache*.

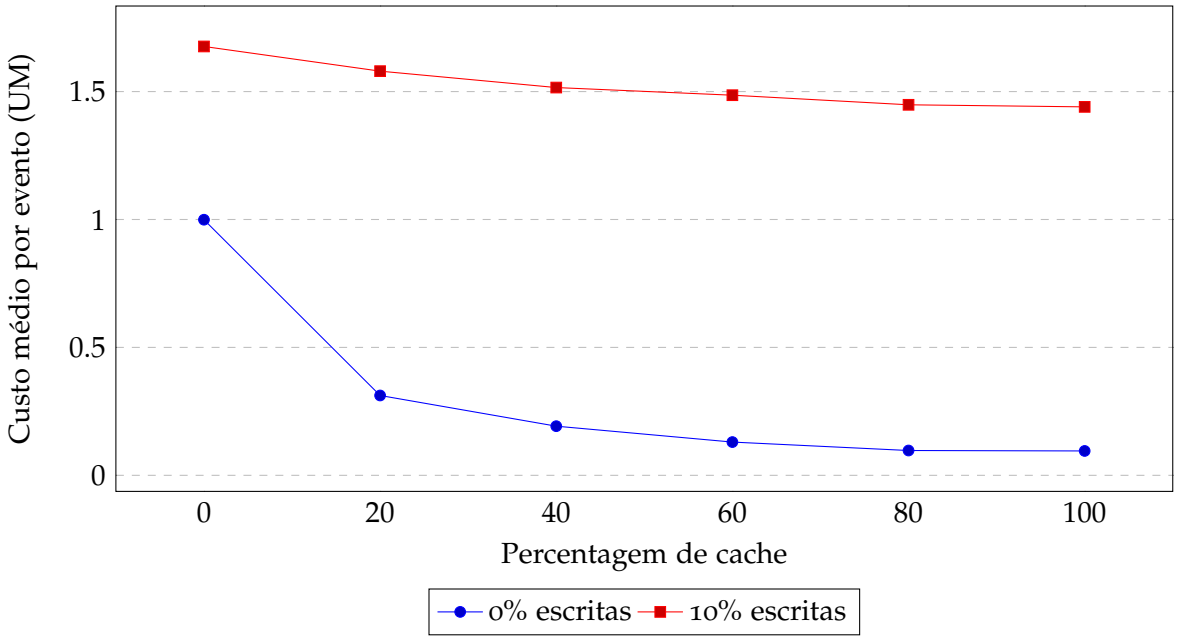


Figura 15: Variação do custo médio por evento, com a variação do tamanho da *cache*.

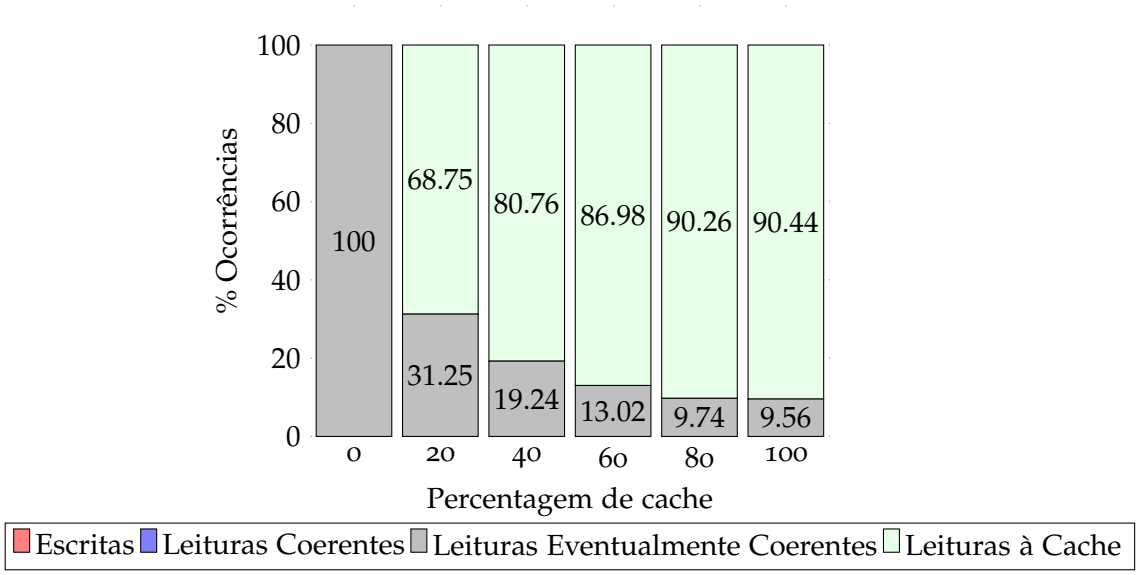


Figura 16: Percentagem de ocorrências de cada operação, com a variação do tamanho da *cache*, com 100% de leituras.

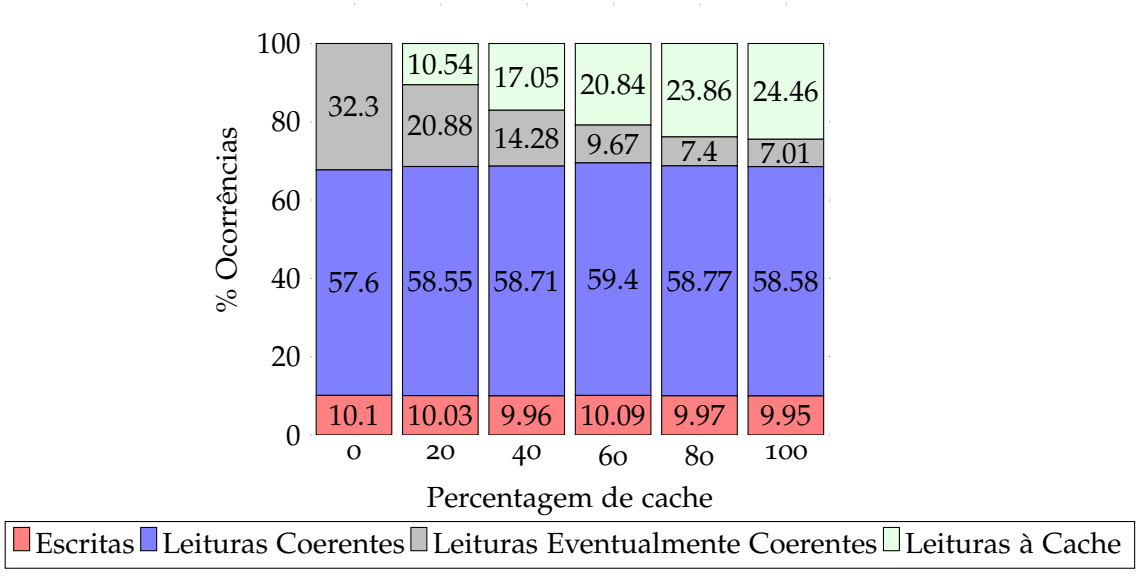


Figura 17: Percentagem de ocorrências de cada operação, com a variação do tamanho da *cache*, com 90% de leituras e 10% de escritas.

5.3.2 *Tempo para a marcação de um item como modificado*

A *workload* utilizada foi a seguinte:

```
Número de runs: 1;
Número de clientes: 10;
Número de chaves: 1000;
```

```
Número de eventos: 10000;
Percentagem de operações de leitura: 90%;
Percentagem de operações de escrita: 10%;
Intervalo entre limpezas: 30 ms;
Tempo para efetuar uma escrita: 44 ms;
Tempo para efetuar uma leitura coerente: 44 ms;
Tempo para efetuar uma leitura eventualmente coerente: 16 ms;
```

Excerto 5.4: *Workload* para a simulação da marcação de um item como modificado.

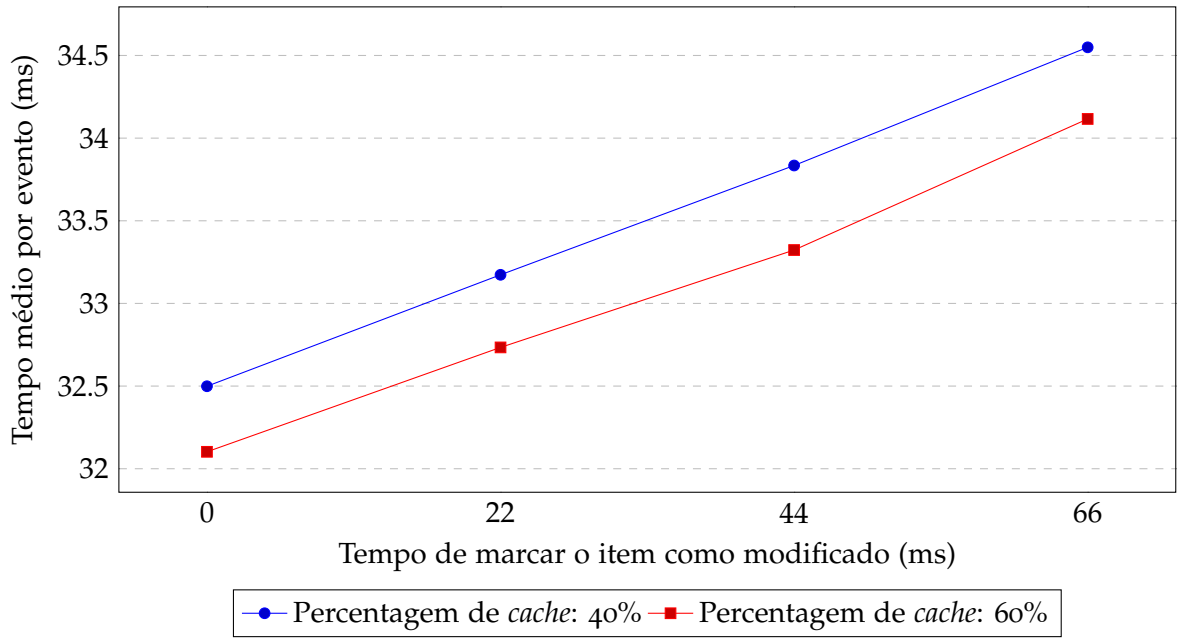


Figura 18: Variação do tempo médio por evento, com a variação do tempo de marcação de um item como modificado.

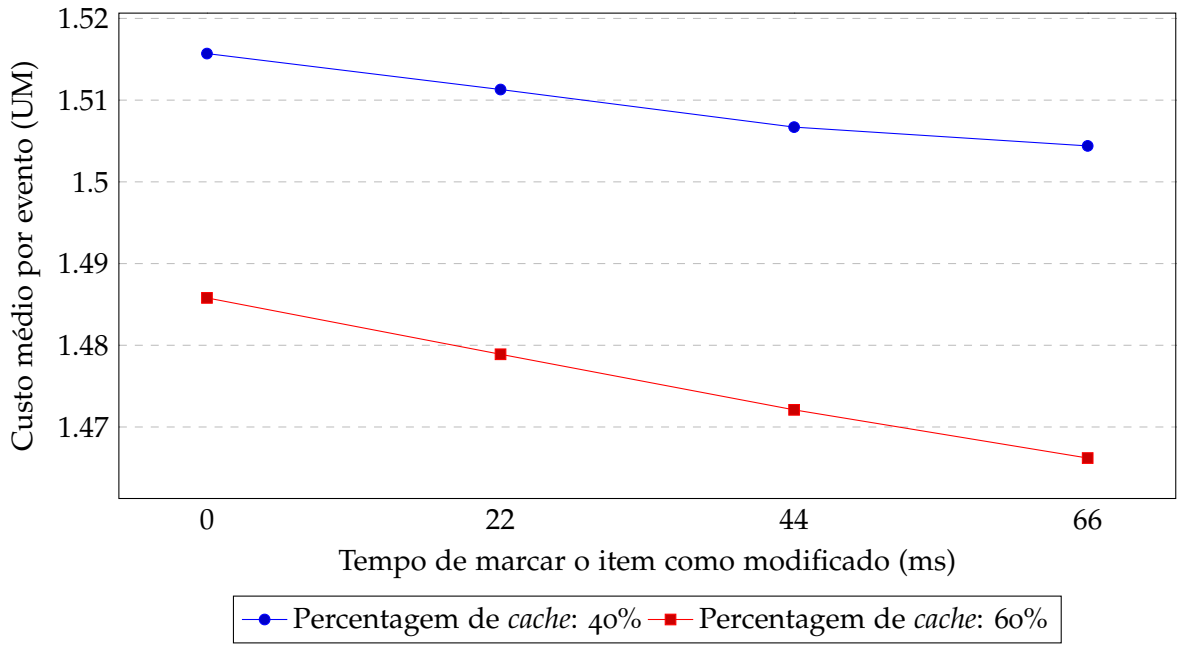


Figura 19: Variação do custo médio por evento, com a variação do tempo de marcação de um item como modificado.

Tal como apresentado na secção anterior, 5.2.2, a marcação de uma chave como alterada após uma escrita tem impacto na performance em questões de tempo gasto na execução das operações (gráfico 18), no entanto, esse tempo gasto poderá ser traduzido numa redução de custos, tal como a figura 19 indica.

5.4 DISCUSSÃO

Os resultados obtidos mostram, em primeiro lugar, que a utilização de mecanismos de *cache* permite, simultaneamente, reduzir o custo e aumentar o desempenho das operações num serviço de armazenamento na nuvem, mesmo com uma percentagem moderada de operações de escrita, tal como demonstram os gráficos 8, 9, 14 e 15.

Por outro lado, como seria de esperar, verifica-se que o desempenho da solução proposta piora quando se assume um mecanismo de coordenação usado na camada de *middleware* para marcar items de dados como modificados. O modelo proposto serve assim para avaliar qual o desempenho mínimo para um mecanismo de coordenação, de forma a satisfazer os requisitos de uma aplicação em particular.

6

CONCLUSÕES

Com o crescimento evidente da utilização de serviços de armazenamento na nuvem, é fundamental que haja um estudo sobre como serviços deste tipo se comporta, em questões de tempo, desempenho e, obviamente, em termos de custos cobrados aos clientes. Assim, um dos propósitos desta dissertação foi, precisamente, realizar essa análise. Foi constatado que existem diferentes modelos de coerência sobre os quais as operações podem executar. Por exemplo, no caso do Amazon DynamoDB são fornecidos dois modelos para as operações de leituras: coerentes e eventualmente coerentes. Já o Azure Cosmos DB dispõe de cinco modelos diferentes.

Foi proposta uma arquitetura de *middleware* de forma a garantir que os acessos ao serviço de armazenamento, para a execução de operações de leitura e escrita, sejam coerentes sempre que necessário, assegurando assim que os clientes apenas pagarão o mínimo possível e necessário. Para tal faz-se uso de uma *cache* local, em cada um dos clientes, e também da marcação de uma chave como alterada, aquando de uma escrita.

Esta arquitetura foi avaliada com a utilização de simulação. Embora um simulador seja apenas uma representação aproximada da realidade, que pode dar lugar a algumas incertezas nos resultados, tentou-se, ao máximo, retratar as situações que poderiam ocorrer num contexto real. Para tal, foram efetuadas diversas experiências, simulando, por exemplo, a latência das operações de leitura e escrita, algo que é impossível de contornar, aquando da utilização de um serviço de armazenamento na nuvem.

Acerca dos resultados obtidos, com as experiências realizadas, pode concluir-se que a utilização da *cache* traz uma vantagem significativa, principalmente, no que concerne à poupança a nível de custos monetários cobrados ao cliente. Com a marcação de alteração num determinado item pode concluir-se que abdicando de um algum tempo, se consegue ganhar margem em termos monetários, pagando apenas o necessário.

6.1 TRABALHO FUTURO

Com o resultado obtido neste trabalho, é agora possível considerar o desenvolvimento de uma camada de *middleware* como a proposta aqui, que possa de facto realizar os benefícios previstos e servir de base para uma avaliação experimental detalhada.

O simulador aqui proposto poderá ser ainda importante para ajudar na tomada de diferentes decisões, por exemplo, na forma como é feita a limpeza da *cache*, para que a chave a retirar seja aquela que está há mais tempo na *cache* sem ser consultada. De momento, a chave a retirar, quando é necessário inserir uma nova e a *cache* se encontra cheia é escolhida de forma aleatória.

Mais ainda seria interessante estudar a aplicabilidade, do *middleware* elaborado, noutros serviços de armazenamento na nuvem, que ofereçam diferentes níveis de coerência, salvo pequenas alterações, que se vissem ser necessárias. Um exemplo seria o Azure Cosmos DB que oferece um conjunto variado de modelos de coerência.

BIBLIOGRAFIA

Amazon. Disponível em https://aws.amazon.com/. Acedido a 2017-10-07.

AWS Documentation. Disponível em http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html, 2012-08-10. Acedido a 2017-10-15.

Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. In *SIAM Journal on Computing*, pages 593–602, 1994.

Azure Cosmos DB. Disponível em https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, 2017. Acedido a 2017-10-20.

Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10715-5.

Eric A. Brewer. Towards Robust Distributed Systems. In *Symposium on Principles of Distributed Computing (PODC)*, 2000. URL http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

Thomas M. Connolly and Carolyn E. Begg. *DataBase Systems: A Practical Approach to Design, Implementation and Management (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004. ISBN 0321294017.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

Amazon DynamoDB. Disponível em https://aws.amazon.com/pt/dynamodb/, 2012. Acedido a 2017-09-27.

D. Gómez Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th International Conference on Data Engineering*, pages 676–687, March 2014. doi: 10.1109/ICDE.2014.6816691.

Google. Disponível em https://cloud.google.com/. Acedido a 2017-10-07.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15 Issue 4:287 – 317, 1983.

M. Klems, D. Bermbach, and R. Weinert. A Runtime Quality Measurement Framework for Cloud Database Service Systems. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pages 38–46, Sept 2012. doi: 10.1109/QUATIC.2012.17.

Donald Kossmann, Tim Kraska, and Simon Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807231. URL http://doi.acm.org/10.1145/1807167.1807231.

Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687657. URL http://dx.doi.org/10.14778/1687627.1687657.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.229904. URL http://dx.doi.org/10.1109/TSE.1977.229904.

J. J. Levandoski, D. Lomet, M. F. Mokbel, and K. Keliang Zhao. Deuteronomy: Transaction Support for Cloud Data. *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387906.

Microsoft. Disponível em https://azure.microsoft.com/. Acedido a 2017-10-07.

Ngoc Thanh Nguyen. Consensus system for solving conflicts in distributed systems. *Information Sciences*, 147(1–4):91 – 122, 2002. ISSN 0020-0255. doi: http://dx.doi.org/10.1016/S0020-0255(02)00260-8. URL http://www.sciencedirect.com/science/article/pii/S0020025502002608.

Alok Sinha. Client-server Computing. *Commun. ACM*, 35(7):77–98, July 1992. ISSN 0001-0782. doi: 10.1145/129902.129908. URL http://doi.acm.org/10.1145/129902.129908.

Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009. ISSN 0001-0782. doi: 10.1145/1435417.1435432. URL http://doi.acm.org/10.1145/1435417.1435432.

YCSB - DynamoDB Binding. Disponível em https://github.com/brianfrankcooper/YCSB/tree/master/dynamodb, 2015. Acedido a 2017-10-25.

Wei Zhou, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *IEEE Trans. Serv. Comput.*, 5(4):525–539, January 2012. ISSN 1939-1374. doi: 10.1109/TSC.2011.18. URL http://dx.doi.org/10.1109/TSC.2011.18.