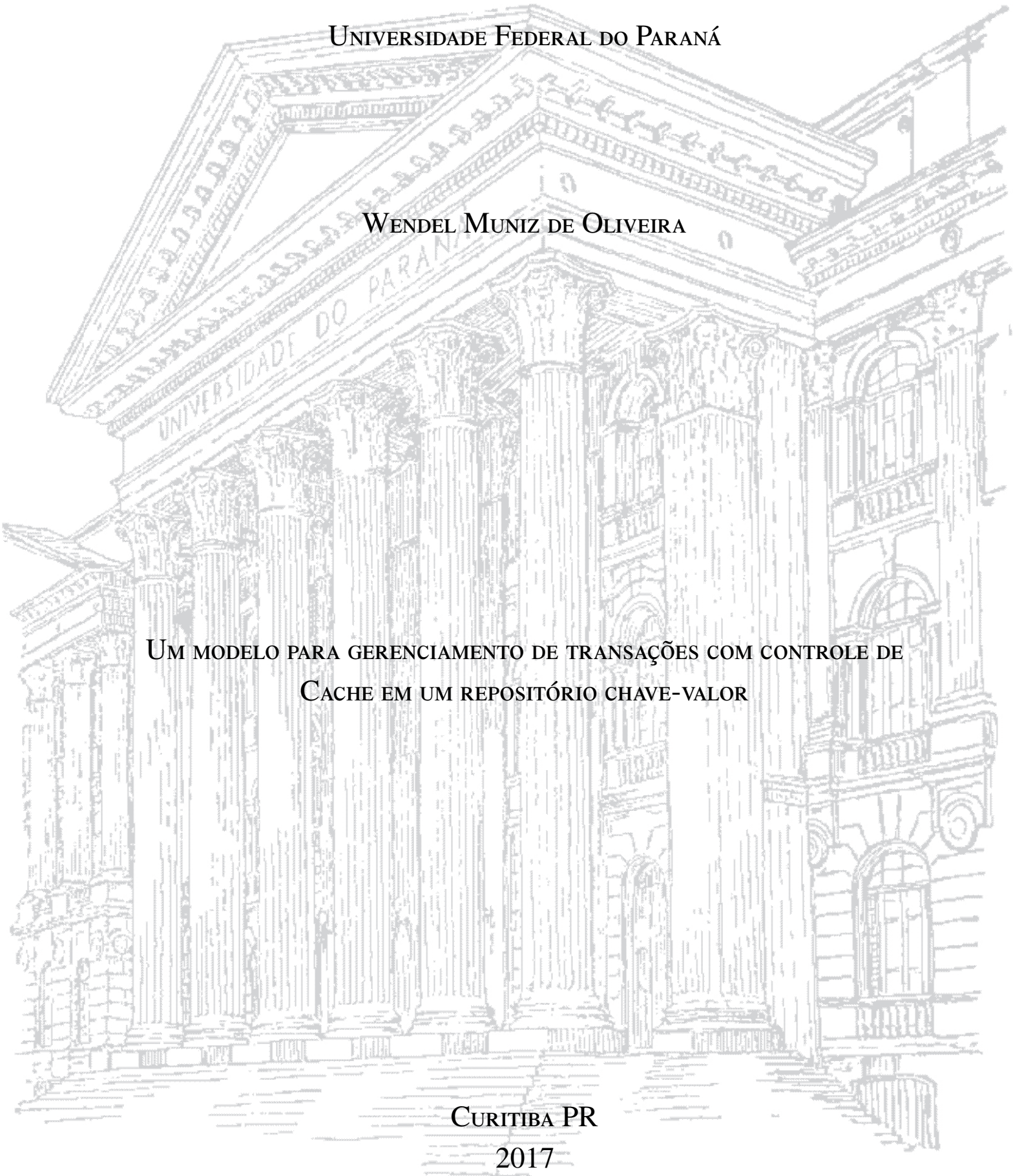
Universidade Federal do Paraná

Wendel Muniz de Oliveira



Um modelo para gerenciamento de transações com controle de Cache em um repositório chave-valor

Curitiba PR
2017

Wendel Muniz de Oliveira

Um modelo para gerenciamento de transações com controle de Cache em um repositório chave-valor

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática, no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Profa. Dra. Carmem Satie Hara.

Curitiba PR
2017

O48m Oliveira, Wendel Muniz de
Um modelo para gerenciamento de transações com controle de cache em um repositório chave-valor / Wendel Muniz de Oliveira. – Curitiba, 2017.
57 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Carmem Satie Hara .
Bibliografia: p. 47-49.

1. Banco de dados – Gerência. 2. Sistemas de recuperação da informação. 3. Gerenciamento de memória (Computação). 4. Memória cache. I. Universidade Federal do Paraná. II.Hara, Carmem Satie. III. Título.

CDD: 005.74

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **WENDEL MUNIZ DE OLIVEIRA** intitulada: **Um Modelo para Gerenciamento de Transações com Controle de Cache em um Repositório Chave-Valor**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 14 de Setembro de 2017.

Carmem Satie Hara

CARMEM SATIE HARA
Presidente da Banca Examinadora (UFPR)

Flávio Rubens de Carvalho Sousa

FLÁVIO RUBENS DE CARVALHO SOUSA
Avaliador Externo (UFC)

LUIZ CELSO GOMES JUNIOR
Avaliador Externo (UTFPR)

Este trabalho é dedicado a minha amada esposa Shirley Tehlen Scheibner cujo apoio foi essencial durante os anos de estudo.

Agradecimentos

Agradeço a Deus por me dar sabedoria e renovar as minhas forças durante a produção deste trabalho. A Diretoria de Gestão de Tecnologia da Informação, departamento no qual trabalho na UTFPR por permitir que me ausentasse por um período para completar o curso. À minha colega e professora Raquel Kolitski Stasiu pelo apoio e conselhos. À professora Carmem pelos ensinamentos e orientação. À minha família por compreender os períodos de ausência e motivação nos momentos difíceis. Finalmente agradeço ao pessoal do LBD pelo apoio e amizade.

Resumo

As estratégias mais comuns para alocação de dados em sistemas distribuídos são as tabelas de dispersão distribuídas (DHT) e os sistemas de diretórios distribuídos. As DHTs garantem escalabilidade, porém não dão às aplicações usuárias controle sobre a localidade dos dados. Por outro lado, os diretórios distribuídos mantêm o mapeamento entre os itens alocados e os servidores que compõem o sistema, o que garante flexibilidade de alocação, mas com escalabilidade limitada. Em um Sistema Gerenciador de Banco de Dados (SGBD), o controle sobre a localidade pode garantir a proximidade dos dados que são frequentemente acessados de forma conjunta nas consultas, com o intuito de reduzir acessos remotos que aumentam o tempo de execução. O ALOCS é um sistema desenvolvido sobre diretórios distribuídos que tem por finalidade ser utilizado como *backend* de armazenamento de um SGBD. Ele adota o conceito de *buckets*, compostos por um conjunto de pares chave-valor, como unidade de comunicação de dados entre servidores. Dessa forma, a aplicação usuária pode alocar em um mesmo *bucket* pares que são frequentemente utilizados em conjunto. Para minimizar ainda mais a quantidade de comunicação, o ALOCS mantém *buckets* previamente acessados em cache. A utilização de cache pode gerar problemas para a consistência dos dados quando vários servidores mantêm em cache *buckets* com dados atualizados.

O objetivo desta dissertação é desenvolver uma solução para manter a consistência entre os dados atualizados em cache e o sistema de armazenamento distribuído. A solução é baseada no modelo de concorrência multiversão, com transações que garantem o isolamento por *snapshot*. Ele foi escolhido por sua abordagem otimista e por não bloquear transações somente de leitura. O sistema foi implementado e os experimentos mostram o impacto da alocação de dados sobre o desempenho do sistema, bem como o *overhead* do protocolo de controle de concorrência sobre o tempo de recuperação e escrita de dados.

Os resultados demonstraram a importância do controle sobre a localidade dos dados. O uso do cache foi determinante para reduzir o tempo de execução das consultas.

Palavras-chave: controle de concorrência, controle de localidade, cache.

Abstract

The most common strategies for data allocating in distributed systems are Distributed Hash Tables (DHT) and Distributed Directory Systems. DHTs guarantee scalability but do not allow control over data location to user applications. On the other hand, distributed directories store the location of data items, that is, a mapping between the stored data and servers that compose the system. This strategy guarantees flexibility of allocation but limits its scalability. In a Database Management Systems (DBMS), control over data locality can ensure the proximity of data that are frequently accessed together in queries in order to reduce the number of remote accesses that increase their execution time. ALOCS is a system developed on distributed directories to be used as a storage backend for DBMSs. It adopts the concept of buckets, composed by a set of key-value pairs, as the communication unit between servers. In this way, the user application can allocate pairs that are often used together in the same bucket. To further minimize the amount of communication, ALOCS maintains previously accessed buckets in cache. Caching can cause problems for data consistency when multiple servers cache buckets with updated data.

The main objective of this dissertation is to develop a solution to maintain the consistency of the updated data in the cache and the storage system. The solution is based on a multiversion concurrency control with snapshot isolation. It has been chosen for its optimistic approach and non-blocking read-only transactions. The system was implemented and our experiments show the impact of data allocation on the system performance as well as the overhead of the concurrency control protocol on the data recovery and writing time.

The results show the importance of allocation control on reducing the execution time of queries. Moreover, they show that caching is crucial to reduce the query execution time.

Keywords: concurrency control, locality control, cache.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

ACID	*Atomicity, Consistency, Isolation, Durability*
ANSI SQL	*American National Standards Institute Structured Query Language*
CAP	*Consistency, Availability, Partition*
MVTO	*Multiversion Timestamp Ordering*
NTP	*Network Time Protocol*
NoSQL	*Not Only SQL*
PCSI	*Partitioned Causal Snapshot Isolation*
PSI	*Parallel Snapshot Isolation*
2V2PL	*Two Version Two-phase locking*

Capítulo 1

Introdução

O volume de dados produzido por sistemas que trafegam dados na internet, como redes sociais e dados coletados de sensores [Tran, 2013], tem crescido muito nos últimos anos [Gantz e Reinsel, 2012]. Além disso, cresceu também a quantidade de acessos a estes dados. Estes fenômenos acarretaram alguns problemas, tais como o aumento da latência de rede para o acesso aos dados e a falta de espaço para o armazenamento.

A necessidade de obter escalabilidade de processamento e aumento da capacidade de armazenamento incentivou a criação de sistemas de armazenamento distribuído que adotam uma abordagem conhecida como NoSQL [Agrawal et al., 2010]. Tais sistemas possuem um conjunto de operações de leitura e escrita mais simples e possuem maior capacidade de armazenamento. Um modelo de dados muito utilizado nesta abordagem é o chave-valor, por ser simples e flexível [Cattel, 2010].

Com a expansão dos sistemas de armazenamento distribuído, os problemas relacionados à alocação de dados, distribuição e localização, começaram a ser discutidos com maior frequência: [Paiva e Rodrigues, 2015] [Paiva et al., 2014] [Li et al., 2014] [Kumar et al., 2014]. O trabalho de Paiva e Rodrigues [Paiva e Rodrigues, 2015] traz uma discussão sobre o tema, enfatizando o grande impacto que algumas características do sistema tem sobre o seu desempenho, tais como o número de acessos remotos, a latência de rede e o congestionamento da rede.

As estratégias mais comuns para alocação de dados, conforme Paiva e Rodrigues [Paiva e Rodrigues, 2015] são as tabelas de dispersão distribuídas (DHT) e os sistemas de diretórios distribuídos. As DHTs garantem escalabilidade, porém, não são flexíveis. Por outro lado, os diretórios distribuídos mantêm o mapeamento entre os itens alocados e os servidores que compõem o sistema, o que garante flexibilidade de alocação, mas tem escalabilidade limitada. Dentre os sistemas que adotam uma DHT podem ser citados: o Cassandra [Lakshman e Malik, 2010] e Dynamo [DeCandia et al., 2007]. Exemplos de sistemas que adotam diretórios distribuídos incluem: PNUTS [Cooper et al., 2008] e BigTable [Chang et al., 2008]. A DHT não é flexível, pois, a alocação é realizada de forma aleatória, não sendo possível otimizar ou controlar a localidade dos dados. O uso de diretórios não é escalável, pois, o custo para manter o mapeamento é alto, e o tempo para obter a localidade dos dados é maior.

1.1 Motivação e Definição do Problema

O controle sobre a localidade dos dados é importante para garantir a proximidade dos dados que são acessados de forma conjunta durante as consultas, com o intuito de reduzir acessos remotos que aumentam o custo de execução das mesmas [Shute et al., 2013]. Em ambientes

geograficamente distribuídos, a localidade dos dados também pode ser ajustada através de políticas de replicação, como demonstrado por [Corbett et al., 2013].

As técnicas para replicação total e parcial dos dados são utilizadas com sucesso para ajustar a localidade dos dados. Um ponto negativo da replicação total é o custo de comunicação entre os servidores para a sincronização dos dados [Silva et al., 2015]. A replicação parcial como tratado em [Padhye et al., 2014] resolve este problema, pois, os dados não são replicados em todos os servidores do sistema, e a propagação das atualizações é enviada somente para os servidores que contêm os dados atualizados. Um ponto negativo na replicação é sobrecarga do sistema com o aumento da quantidade de leituras remotas, prejudicando o tempo de resposta das consultas.

A localidade dos dados pode ser ajustada também através da alocação dos dados remotos em cache local [Silva et al., 2015]. Uma das estratégias mais utilizadas é manter em cache os dados mais acessados como o proposto em [Silva et al., 2015] e [Sovran et al., 2011]. O sistema proposto em [Corbett et al., 2013] mantém em cache os dados modificados, e periodicamente os escreve em disco. A utilização de cache pode gerar problemas para a consistência dos dados, portanto, é necessário criar um mecanismo para controle de concorrência que garanta a consistência [Silva et al., 2015].

O ALOCS [Bungama et al., 2016] é um sistema desenvolvido pelo Grupo de Gerenciamento de Dados e Informação da UFPR, que explora a alocação de dados com controle sobre a localidade. O objetivo do ALOCS é gerenciar a alocação de dados em sistemas de armazenamento distribuído, promovendo a comunicação entre a aplicação, os sistemas de armazenamento e o controle de metadados, permitindo a alocação de um conjunto de pares chave-valor agrupados em uma única estrutura, cuja localidade é controlada pela aplicação. O ALOCS mantém em cache todos os dados processados tanto em consultas como em atualizações.

1.2 Objetivos e Contribuições

O objetivo principal deste trabalho é desenvolver uma solução para manter a consistência dos dados mantidos em cache e o sistema de armazenamento.

As contribuições deste trabalho são:

- proposta de um modelo de armazenamento multiversão para o ALOCS;
- desenvolver um protocolo para gerenciamento de transações, que garante a consistência de dados mantidos em cache em um sistema de armazenamento distribuído;
- implementação do modelo proposto e um estudo experimental que mostra o impacto da alocação de dados sobre o desempenho do sistema, bem como o *overhead* do protocolo de controle de concorrência sobre o tempo de recuperação e escrita de dados.

1.3 Organização do Trabalho

O restante do documento está organizado da seguinte forma. O capítulo 2 traz as definições preliminares dos conceitos utilizados nesta dissertação. O capítulo 3 descreve alguns trabalhos relacionados ao problema tratado neste trabalho. O capítulo 4 apresenta o ALOCS, como proposto por Bungama [Bungama et al., 2016]. As extensões do ALOCS para o gerenciamento de transações são descritas no capítulo 5. O capítulo 6 descreve o estudo experimental, seguido pelas conclusões no capítulo 7.

Capítulo 2

Definições Preliminares

Este capítulo descreve alguns conceitos importantes relacionados a controle de concorrência e consistência de dados utilizados nesta proposta. A seção 2.1 trata sobre a consistência de dados e a importância da ordenação de eventos em sistemas distribuídos. A seção 2.2 trata sobre o controle de concorrência e a importância desta atividade para os sistemas de banco de dados. A seção 2.2.2 trata sobre o controle de concorrência multiversão e os mecanismos mais utilizados para implementá-lo. A seção 2.2.3 define o isolamento por *snapshot*, que é um mecanismo para controle de concorrência multiversão. A seção 2.2.1 descreve os níveis de isolamento definidos pelo padrão ANSI SQL [ANSI, 1986].

2.1 Consistência de Dados

A consistência de dados nos sistemas de banco de dados é um fator crítico, principalmente quando se consideram as estratégias de replicação [Bravo et al., 2015].

Para garantir a consistência dos dados os sistemas de banco de dados transacionais devem seguir quatro propriedades: atomicidade, consistência, isolamento e durabilidade. Estas propriedades são conhecidas pelo acrônimo ACID.

Em sistemas de armazenamento distribuídos sujeitos à falha, foi comprovado através do teorema CAP que não é possível garantir de forma integral a consistência, disponibilidade, e tolerância ao particionamento. Este teorema estabelece que somente duas das três propriedades podem ser obtidas ao mesmo tempo [Vogels, 2009].

Entre os vários tipos de consistência, destacam-se os seguintes: consistência forte, consistência eventual e consistência causal. A consistência forte corresponde à manutenção das propriedades ACID. A consistência eventual garante que em algum momento todos os servidores que compõem um sistema receberão as atualizações propagadas. A consistência causal é uma variação da consistência eventual e garante que todas as transações que tenham dependência entre as operações sejam persistidas e propagadas juntas [Vogels, 2009].

2.1.1 Ordenação de Eventos

A garantia de consistência na persistência dos dados depende da capacidade dos sistemas em manter a ordem em que as operações de consulta e atualização são realizadas. A execução das operações de atualização na sequência incorreta pode ocasionar anomalias na persistência e provocar inconsistências na base de dados [Bravo et al., 2015].

Um método eficiente para ordenar operações é utilizar relógios para atribuir *timestamps* aos eventos de atualização. Os relógios são utilizados em sistemas distribuídos para resolver

problemas de sincronização através da ordenação de eventos. Quando o relógio mantém um *timestamp* único é conhecido como escalar, e quando mantém um *timestamp* para cada servidor, é conhecido como vetor de relógios [Bravo et al., 2015]. Há dois tipos de relógios: físicos e lógicos. Os relógios físicos são associados a meios externos de controle do tempo [Bravo et al., 2015].

Os relógios lógicos determinam a ordem dos eventos baseados na relação "aconteceu antes de". Esta relação diz que se um evento A ocorreu antes do evento B, então o evento A deve ser atendido primeiro. O meio utilizado para determinar esta ordem é a atribuição de sequências numéricas aos eventos ocorridos em cada processo no sistema [Lamport, 1978].

2.2 Controle de Concorrência

Controle de concorrência é a atividade de coordenar o acesso intercalado aos recursos compartilhados disponibilizados em um sistema. A execução concorrente dos processos que acessam estes recursos podem interferir no resultado obtido pelos usuários do sistema [Bernstein et al., 1987].

Uma transação é um conjunto de operações que acessam e modificam dados no sistema de banco de dados. O objetivo do controle de concorrência é garantir que as transações sejam executadas de forma atômica. A execução atômica significa que uma transação não interfere no resultado de outras transações, e os resultados são permanentes [Bernstein et al., 1987].

Os problemas encontrados no controle de concorrência tornam-se maiores nos cenários em que são utilizados sistemas de banco de dados distribuídos devido aos seguintes aspectos: os usuários podem acessar dados armazenados em diferentes servidores e o processo para controle de concorrência de um servidor não recebe no mesmo instante as mudanças ocorridas nos outros servidores que compõem o sistema [Bernstein e Goodman, 1981].

A teoria da serialização permite analisar a execução de algoritmos para controle de concorrência. O objetivo é verificar se o algoritmo analisado coordena as operações da forma correta. Nesta teoria a execução das transações é representada através de um histórico que detalha a ordem em que as operações de leitura e escrita foram executadas [Bernstein et al., 1987]. Um histórico é considerado serializável quando produz o mesmo resultado de uma execução não concorrente de todas as transações. A execução serializável de transações é considerada correta porque conserva o estado consistente da base de dados após a efetivação das modificações [Bernstein e Goodman, 1981].

A serialização correta das transações está condicionada ao tratamento adequado dos conflitos entre operações concorrentes. Um conflito ocorre quando duas operações acessam o mesmo dado e uma delas é uma escrita. Há dois tipos de conflitos: entre uma operação de leitura e outra de escrita; e entre duas operações de escrita. Os conflitos entre leituras e escritas ocorrem quando uma operação de escrita tenta modificar um dado que está sendo lido por outra operação [Bernstein e Goodman, 1981].

Os mecanismos para controle de concorrência geralmente são baseados nos protocolos *two-phase locking* e *timestamp ordering* para coordenar a execução das transações [Bernstein e Goodman, 1981]. O protocolo *two-phase locking* permite a execução das operações de leitura e escrita, somente após a aquisição de bloqueios apropriados. Para a execução de operações de leitura é necessário a obtenção de um bloqueio compartilhado, que permite outras transações obter bloqueio para leitura no mesmo conjunto de dados. Para a execução de operações de escrita é necessário a aquisição de um bloqueio exclusivo. O protocolo *timestamp ordering* atribui, de forma atômica, *timestamps* às transações. Os *timestamps* permitem a ordenação de operações em conflito para garantir a serialização adequada.

2.2.1 Anomalias na Serialização

A execução intercalada de operações de leitura e escrita concorrentes pode ocasionar inconsistência na base de dados [Bernstein e Goodman, 1983]. Estas inconsistências são causadas por anomalias na persistência, provocadas por conflitos entre operações [Berenson et al., 1995].

Os níveis de isolamento foram estabelecidos para coibir o surgimento das anomalias e permitem fazer um equilíbrio entre o grau de concorrência no processamento das operações e a consistência dos dados persistidos. Quanto menor o nível de consistência, maior será o grau de concorrência. Porém, a base de dados fica sujeita à ocorrência de inconsistências causadas por anomalias na persistência dos dados [Berenson et al., 1995].

O padrão ANSI SQL-92 [ANSI, 1986] define os níveis de isolamento baseado em fenômenos que podem causar anomalias na persistência dos dados [Berenson et al., 1995]. Os fenômenos apresentados pelo padrão estão listados a seguir.

- **Leitura Suja**: Ocorre quando uma transação T1 modifica um dado, uma transação T2 lê o dado modificado por T1, e em seguida T1 é abortada. A transação T2 leu um dado que não foi persistido.
- **Não Repetível**: Ocorre quando uma transação T1 lê um dado, uma transação T2 modifica ou remove o dado lido por T1, e em seguida T1 necessita ler o dado novamente. A transação T1 obterá um resultado diferente da primeira leitura em virtude da execução da transação T2.
- **Fantasma**: Ocorre quando uma transação T1 lê um conjunto de dados que satisfazem uma condição de busca, uma transação T2 insere dados que satisfazem a condição de busca da transação T1, e T1 repete a leitura com a mesma condição de busca. A transação T1 obterá um resultado diferente da primeira leitura em virtude da execução da transação T2.

Baseado nos fenômenos listados acima foram estabelecidos os níveis de isolamento a seguir [Berenson et al., 1995]: *Read Uncommited*, *Read Commited*, *Repeatable Read* e *Anomaly Serializable*. A Tabela 2.1 mostra a relação entre os níveis de isolamento e os fenômenos que podem causar anomalias.

Tabela 2.1: Níveis de Isolamento em relação aos fenômenos.

Nível de Isolamento	**Leitura Suja**	**Não Repetível**	**Fantasma**
Read Uncommited	Pode ocorrer	Pode ocorrer	Pode ocorrer
Read Commited	Evita	Pode ocorrer	Pode ocorrer
Repeatable Read	Evita	Evita	Pode ocorrer
Anomaly Serializable	Evita	Evita	Evita

O nível *Read Uncommited* é considerado o mais fraco, pois permite que todos os fenômenos ocorram. O nível *Anomaly Serializable* representa qualquer nível que garanta a serialização do histórico de execução das transações, e é considerado o mais forte pois evita todas as anomalias. A forma mais comum para obtenção da serialização é a implementação do controle de concorrência baseado no protocolo *two-phase locking* [Berenson et al., 1995], pois requer bloqueios específicos para a execução das operações de leitura e escrita.

2.2.2 Controle de Concorrência Multiversão

Em um sistema de banco de dados multiversão cada dado contém uma ou mais cópias, que são conhecidas como versões. Nestes sistemas as atualizações produzem novas versões dos dados [Bernstein e Goodman, 1983].

No controle de concorrência multiversão as transações recebem no início da execução um *timestamp*. O controle de concorrência permite que sejam lidos somente os dados com versões menores ou iguais aos *timestamps* atribuídos às transações. As versões são identificadas pelos *timestamps* atribuídos às transações que modificaram os dados.

Os mecanismos que implementam o controle de concorrência multiversão geralmente possuem um módulo, denominado escalonador [Bernstein e Goodman, 1981], que é responsável por permitir a execução das operações. Este escalonador deve resolver os conflitos entre operações para garantir a serialização das transações.

No mecanismo *Multiversion Timestamp Ordering* (MVTO) [Bernstein et al., 1987] baseado no protocolo *timestamp ordering*, o acesso aos dados é baseado no *timestamp* que a transação recebe no início da execução. No algoritmo MVTO as operações podem ser processadas fora da ordem dos *timestamps* das transações. Esta condição pode gerar um conflito entre operações de leitura e escrita concorrentes. Para que este conflito não ocorra, o escalonador mantém uma lista das versões para cada item de dado. Nesta lista cada versão é associada a um intervalo de *timestamps* denotado por int(X_i) = {*wts*,*rts*}, onde *wts* é o maior timestamp da transação que escreveu o item de dado X_i, e *rts* é o maior *timestamp* atribuído a uma transação que tenha lido o item de dado X_i.

No mecanismo MVTO, para executar uma leitura o escalonador deve selecionar a versão com o maior *wts* que seja menor ou igual ao *timestamp* da transação. Se este *timestamp* for maior que *rts*, o valor de *rts* é substituído. Para executar uma escrita o escalonador seleciona a versão com o maior *wts* que seja menor que o *timestamp* da operação. Para evitar que uma transação modifique um item de dado que está sendo lido por outra transação, o valor de *rts* do intervalo é verificado. Se o valor de *rts* do intervalo for maior que o *timestamp* da transação que pretende modificar o item de dado, ela deverá ser interrompida pois, há uma transação lendo o item de dado. Se a transação não for interrompida uma nova versão do dado é criada com o timestamp recebido no início da execução.

No mecanismo *Two Version Two-phase locking* (2V2PL) [Bernstein et al., 1987] baseado no protocolo *two-phase locking*, o acesso aos dados é baseado na aquisição de bloqueios sobre os dados para a execução das operações. Para executar uma leitura, após a aquisição do bloqueio apropriado o escalonador seleciona a versão persistida com o maior *timestamp*. É possível ler dados que tenham sido escritos por operações de escrita pertencentes a mesma transação. Para executar uma escrita, após a aquisição do bloqueio apropriado, o escalonador cria uma versão com o *timestamp* recebido pela operação de escrita. Os bloqueios de escrita e leitura são compatíveis entre si. O escalonador sempre seleciona a versão persistida com o maior *timestamp* para a leitura. Se o dado a ser lido estiver com um bloqueio para escrita, o escalonador seleciona uma versão anterior. Desta forma as operações de leitura não são atrasadas por conta dos bloqueios de escrita nos dados.

Há um mecanismo, proposto em [DuBourdieux, 1982], para implementação do controle de concorrência multiversão que mescla os protocolos *timestamp ordering* e *two-phase locking*. Neste mecanismo as transações somente leitura são classificadas como **consultas**, e as que modificam dados como **atualizações**. O escalonador deve classificar as transações antes de executá-las. Para a execução das consultas é utilizado o mecanismo MVTO, e para a execução das atualizações é utilizado o protocolo *Strict 2PL* [Bernstein et al., 1987]. As transações de consulta recebem um *timestamp* menor do que qualquer *timestamp* que tenha sido atribuído

para uma transação de atualização ativa. Esta regra serve para evitar que uma consulta obtenha dados não persistidos. As transações de atualização recebem um *timestamp* após as versões serem escritas, no momento em que forem persistidas. O benefício deste mecanismo é o aumento da concorrência entre transações, pois, não há atrasos de operações em decorrência da espera para obter os bloqueios necessários como ocorre no 2V2PL [Bernstein et al., 1987]. A desvantagem é a necessidade de manter uma lista com versões persistidas para auxiliar na seleção de versões [Bernstein et al., 1987], o que pode causar uma sobrecarga no desempenho do sistema no processamento das consultas.

2.2.3 Isolamento por *Snapshot*

O isolamento por *snapshot* foi definido por Berenson et al. [Berenson et al., 1995]. É baseado no mecanismo que mescla os protocolos *timestamp ordering* e *two-phase locking* descritos na seção 2.2.2. A vantagem deste mecanismo, demonstrado em [Berenson et al., 1995], é a imposição de um nível de isolamento reduzido possibilitando um alto grau de concorrência entre operações, sem permitir a maioria das anomalias descritas na seção 2.2.1.

No controle de concorrência baseado no isolamento por *snapshot* a execução das operações é baseada apenas no *timestamp* que a transação recebe no início da execução. A diferença entre o isolamento por *snapshot* e os mecanismos descritos na seção 2.2.2, é que o isolamento por snapshot permite que uma transação modifique um dado que está sendo lido por outra transação, mas a transação que está lendo o dado modificado tem acesso somente à versão que foi persistida antes da transação iniciar. Esta regra proíbe a ocorrência da anomalia de leitura suja descrita na seção 2.2.1. Quando a transação é encerrada recebe um *timestamp* de *commit*. Duas transações são consideradas concorrentes quando o intervalo entre os *timestamps* de início e *commit* das transações se sobrepõem [Fekete et al., 2005].

Para executar as operações o escalonador seleciona as versões com *timestamp* menor ou igual à atribuída à transação. Como ocorre no MVTO as transações somente de leitura não são bloqueadas por transações concorrentes.

As transações que envolvem operações de escrita devem passar por uma fase de validação. Na validação o escalonador segue uma regra denominada *First Commiter Wins* [Fekete et al., 2005]. A *First Commiter Wins* determina que uma transação T_i só poderá persistir suas modificações se nenhuma outra transação concorrente persistiu modificações no mesmo conjunto de dados. Há uma variação desta regra, utilizada em uma implementação da Oracle [Jacobs et al., 1995] conhecida como *First Updater Wins* [Fekete et al., 2005]. A *First Updater Wins* é baseada na aquisição de bloqueio para a conclusão das operações. Quando duas transações concorrentes T_i e T_j necessitarem modificar o mesmo conjunto de dados, a transação que obtiver o bloqueio primeiro conseguirá persistir as modificações. Se a transação T_i obter o bloqueio primeiro, T_j não conseguirá acessar os dados para modificação e deverá ser interrompida. Se T_i estiver ativa quando T_j solicitar o bloqueio, T_j será interrompida somente após o término de T_i. Se, por outro lado, T_i já tiver sido concluída, T_j será interrompida no mesmo instante. A transação T_j conseguirá adquirir o bloqueio somente se T_i sofrer uma interrupção.

2.3 Sumário

Este capítulo apresentou o controle de concorrência multiversão. No controle de concorrência multiversão as transações recebem no início da execução um *timestamp*. O controle de concorrência permite que sejam lidos somente os dados com versões menores ou iguais aos *timestamps* atribuídos às transações. As versões são identificadas pelos *timestamps* atribuídos às

transações que modificaram os dados. No controle de concorrência multiversão as atualizações produzem novas versões dos dados.

Neste capítulo são apresentados quatro mecanismos para a implementação do controle de concorrência multiversão: *Multiversion Timestamp Ordering* (MVTO), *Two Version Two-phase locking* (2V2PL), isolamento por *snapshot* e um mecanismo que mescla os protocolos utilizados no MVTO e 2V2PL.

O MVTO é baseado no protocolo *timestamp ordering* em que o acesso aos dados é realizado a partir do *timestamp* que a transação recebe no início da execução. O 2V2PL é baseado no protocolo *two-phase locking* em que o acesso aos dados é realizado a partir da aquisição de bloqueios sobre os dados para a execução das operações. A vantagem do MVTO sobre o 2V2PL é não necessitar de bloqueio para as operações de leitura, condição que elimina o tempo de espera pelo bloqueio que há no 2V2PL.

O benefício do mecanismo que mescla os protocolos *timestamp ordering* e *two-phase locking* é o aumento da concorrência entre transações, pois, não há atrasos de operações em decorrência da espera para obter os bloqueios necessários como ocorre no 2V2PL. A desvantagem é a necessidade de manter uma lista com versões persistidas para auxiliar na seleção de versões, o que pode causar uma sobrecarga no desempenho do sistema no processamento das consultas.

O isolamento por *snapshot* é baseado no mecanismo que mescla os protocolos *timestamp ordering* e *two-phase locking*. A vantagem deste mecanismo, demonstrado em [Berenson et al., 1995], é a imposição de um nível de isolamento reduzido possibilitando um alto grau de concorrência entre operações, sem permitir a maioria das anomalias descritas na seção 2.2.1. Esta vantagem influenciou a adoção por este mecanismo para a implementação do protocolo para controle de concorrência no ALOCS apresentado no capítulo 5.

No próximo capítulo serão apresentados alguns trabalhos que tratam sobre o gerenciamento de transações em sistemas de armazenamento NoSql utilizando o modelo multiversão e o controle de concorrência baseado no isolamento por *snapshot*.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados alguns trabalhos que tratam sobre o gerenciamento de transações em sistemas de armazenamento NoSql utilizando o modelo multiversão e o controle de concorrência baseado no isolamento por *snapshot*.

3.1 Padhye e Tripathi

Vinit Padhye e Anand Tripathi em [Padhye e Tripathi, 2015] apontam algumas questões importantes que devem ser consideradas na implementação de um mecanismo para controle de concorrência baseado no isolamento por *snapshot* para sistemas de armazenamento NoSQL e descrevem abordagens que podem ser utilizadas para tratá-las. Além disso, são discutidas possíveis anomalias que podem surgir por problemas na serialização das transações e são propostas soluções para estes problemas.

Os autores propõem um modelo para gerenciamento de transações com suporte à detecção de falhas e um protocolo baseado neste modelo. O protocolo utiliza um serviço centralizado para geração dos *timestamps* de início e *commit*. Para prevenir anomalias na serialização o protocolo exige a aquisição de bloqueios para as operações de leitura e escrita na fase de validação. O bloqueio para operações de leitura é compartilhado, o que possibilita a aquisição de bloqueios para leitura em dados já bloqueados para leitura por outras transações. O modelo exige que o protocolo mantenha uma tabela com metadados sobre as transações em execução. Nesta tabela são mantidas as informações para controle dos bloqueios, as relações de dependência entre transações utilizadas pelo método de prevenção de anomalias e os dados gerados pelas transações para o processo de recuperação.

O método para prevenir anomalias exige que o protocolo na fase de validação identifique conflitos entre operações de leitura e escrita. Ao validar uma transação com operações de escrita, o protocolo verifica se algum dado que está sendo modificado contém bloqueio para operações de leitura. Se houver, a transação que está sendo validada deve ser interrompida.

3.2 PCSI

Padhye et al. [Padhye et al., 2014] tratam sobre o problema de fornecer um mecanismo para controle de concorrência em sistemas de banco de dados parcialmente replicados. A motivação é obter um mecanismo que faça a propagação de atualizações de forma assíncrona e consistente. Os autores propõem um modelo, denominado PCSI, para o gerenciamento de transações baseado no isolamento por *snapshot* e um protocolo que implementa o modelo.

O modelo de sistema que serviu de base para a criação do modelo é um sistema de banco de dados formado por múltiplas partições replicadas em um ou mais servidores. Um servidor pode conter várias partições.

O modelo PCSI garante a ordem causal das transações. A ordem causal ordena duas transações T_i e T_j da seguinte forma: se T_i ler dados modificados por T_j, então T_i precede T_j.

Para a execução das transações o protocolo gera um *snapshot* global que atende as propriedades de atomicidade e ordem causal. Um *snapshot* atende a propriedade de atomicidade quando todas as atualizações ocorridas nos servidores envolvidos forem visíveis. Ele atende a propriedade de ordem causal quando todas as transações obedecem à precedência causal. O *snapshot* global é um vetor composto por subvetores, onde cada subvetor corresponde ao *snapshot* gerado para o servidor S_i envolvido na transação.

No protocolo PCSI o gerenciamento de transações é distribuído. As transações são controlados por partição. Um servidor mantém um contador associado a cada partição local, ele é utilizado para gerar os identificadores das transações. Um *snapshot* contém os *timestamps* de *commit* gerados para as últimas transações que atualizaram dados no servidor S_i.

Ao executar uma transação que envolve múltiplos servidores, o servidor que iniciou o protocolo solicita *snapshot* local aos servidores remotos para gerar o *snapshot* global. Para executar as operações de leitura a transação localiza nos logs uma versão que seja visível ao *snapshot*. Para uma versão ser visível ela deve ser menor ou igual ao *timestamp* no *snapshot*. As operações de escrita são executadas localmente e os dados modificados são mantidos em um *buffer* até a fase de *commit*.

A fase de validação é realizada apenas para as transações que modificam dados por meio do protocolo *two-phase commit*. Na primeira fase o servidor que executou a transação envia uma mensagem de preparação para os servidores remotos com os dados que foram modificados e o *snapshot* utilizado pela transação. Cada participante verifica se a versão modificada está de acordo com o *snapshot*, e a existência de bloqueios para os dados que estão sendo modificados. Se a transação for aprovada na validação o protocolo faz o cálculo das dependências da transação. As dependências da transação são gravadas em um vetor que é enviado aos servidores remotos na propagação das atualizações. O protocolo utiliza o vetor de dependências para garantir a consistência causal das transações. Na fase de *commit* a transação recebe um vetor de *timestamps* de *commit*. Cada posição do vetor corresponde à partição P_i modificada na transação. Os dados modificados são persistidos em logs. Após o *commit* o servidor que executou a transação faz a propagação das atualizações.

Este protocolo foi utilizado como inspiração para a implementação do protocolo para o ALOCS. As características deste protocolo que são semelhantes ao proposto para o ALOCS são: a utilização do isolamento por *snapshot*, a utilização do vetor de relógios e não bloquear as operações somente de leitura. As características que diferenciam o PCSI, do protocolo proposto para o ALOCS são: não utilizar cache local para as operações; e os processos durante a validação das atualizações para manter a consistência causal.

3.3 *Clock-SI*

O *Clock-SI* [Du et al., 2013] é um mecanismo para controle de concorrência para repositórios chave-valor particionados baseado no isolamento por *snapshot*. Cada servidor mantém um relógio físico local para a ordenar as transações. Os relógios são sincronizados pelo protocolo NTP [Mills, 1991]. O *Clock-SI* não utiliza serviço de bloqueio, a consistência é garantida através dos *timestamps* obtidos dos relógios.

Os clientes conectam-se a servidores selecionados por um balanceador de carga para executar transações. Estes servidores são responsáveis por atribuir os *timestamps* de início e *commit* para as transações.

Para ler dados, o *Clock-SI* verifica se o *timestamp* de início é maior que o *timestamp* de *commit* da transação que leu o dado. Se o dado a ser lido estiver sendo atualizado por uma transação com o estado *committing* e o *timestamp* de inicio for maior que o *timestamp* de *commit*, o *Clock-SI* impõe um tempo de espera até que o dado seja persistido. Nas transações remotas, se o *timestamp* de início for maior que o tempo do relógio remoto, o *Clock-SI* impõe um tempo de espera até que tempo do relógio remoto seja maior. *O Clock-SI* retorna para o cliente a maior versão que tenha sido criada antes do *timestamp* de inicio. Os tempos de espera impostos são para garantir a consistência nas transações. Os tempos de espera não causam *deadlocks* pois, os tempos de espera são limitados.

As atualizações de dados são executadas em áreas de trabalho nos servidores que iniciaram a transação. O protocolo *two-phase commit* é utilizado para coordenar a fase de *commit* das transações que envolvem múltiplos servidores. O servidor que iniciou a transação é eleito o coordenador do protocolo. Na primeira fase o coordenador envia aos participantes uma mensagem de preparação contendo os dados que foram modificados. Os participantes executam um processo de validação. Se as modificações forem aprovadas, os partipantes retornam para o coordenador um *timestamp* de preparação. Após receber as respostas de preparação, o coordenador inicia a segunda fase com o envio da mensagem de *commit* com o maior *timestamp* recebido. O *timestamp* enviado pelo coordenador é utilizado como *timestamp* de *commit*.

3.4 *Spanner*

O *Spanner* [Corbett et al., 2013] foi desenvolvido pela *Google* para gerenciar os serviços de replicação de dados entre servidores remotos. Na arquitetura do *Spanner* cada servidor é responsável por instâncias de *tablets*. Uma *tablet* é uma estrutura de dados onde são alocados pares chave-valor. Acima do mapeamento de chaves há uma unidade de replicação e movimentação denominada diretório. Os Diretórios são utilizados para controlar a localização das chaves.

O *Spanner* gerencia a replicação através do protocolo *Paxos* [Lamport, 1998]. Os servidores são organizados em grupos e cada grupo possui um líder. O líder *Paxos* mantém o mecanismo para controle de concorrência sendo o responsável por coordenar a execução do protocolo *two-phase commit* para gerenciar as transações distribuídas. O protocolo *two-phase commit* é executado somente quando a transação envolve mais de um grupo.

O *Spanner* contém uma estrutura de dados denominada *TrueTime* utilizada para obter os *timestamps* de início e *commit* para a execução das transações, que são gerados por um relógio físico. O *Spanner* utiliza a estrutura de dados *TrueTime* e o protocolo *Paxos* para garantir a ordenação total das transações dar suporte à replicação síncrona.

O controle de concorrência do *Spanner* suporta os seguintes tipos de transações: *read-write transaction*, *read-only transaction* e *snapshot reads*. O *Spanner* utiliza mecanismos diferentes de acordo com o tipo de transação. Para as transações que modificam dados utiliza um mecanismo baseado no protocolo *two-phase locking*. Para as outras, ele utiliza um mecanismo baseado no isolamento por *snapshot* com o protocolo *timestamp ordering*.

3.5 *Walter*

O *Walter* [Sovran et al., 2011] é um repositório chave-valor que permite replicação entre servidores distantes de forma assíncrona. É proposto um nível de isolamento baseado no isolamento por *snapshot* denominado *PSI*. O *PSI* permite que as atualizações sejam propagadas para os servidores réplicas sem exigir a ordenação total das transações. Para reduzir o tempo de resposta, o *Walter* coloca em cache os objetos mais acessados.

O *Walter* associa os pares chave-valor a objetos. Os objetos são armazenados em contêineres. Um contêiner é uma unidade de armazenamento lógica que agrupa objetos que tenham um propósito comum. Todos os objetos de um contêiner são armazenados no mesmo servidor, conhecido como *site* preferencial. O Walter prioriza a execução de atualizações nos servidores preferenciais para que as operações sejam menos custosas. Se as atualizações não puderem ser realizadas nos servidores preferenciais ele utiliza um protocolo baseado no *two-phase commit* para persistir os dados.

O *Walter* mantém em cada servidor um histórico de atualizações associadas aos objetos armazenados para a execução das transações. No início da execução a transação recebe um vetor de *timestamps* contendo os identificadores das últimas transações que atualizaram dados no servidor. Os objetos modificados são mantidos em um *buffer* temporário. Para executar as operações de leitura em um servidor, o *Walter* busca a última atualização do objeto no *buffer*. Se não houver atualizações no *buffer* ele identifica uma versão que esteja visível no histórico. Se os dados não estiverem disponíveis localmente, o *Walter* recupera os dados dos servidores preferenciais e os adiciona no histórico. Para uma versão estar visível no histórico, el deve ter o identificador menor igual ao identificador correspondente no vetor de *timestamps*. As operações de escrita são mantidas no *buffer* temporário local até que os dados sejam propagados e persistidos nas réplicas.

Na fase de *commit* o *Walter* faz duas verificações: se o dado a ser modificado foi alterado durante a execução da transação e se existe um bloqueio associado ao dado. Se uma dessas condições for verdadeira a transação é abortada. Ele utiliza protocolos de *commit* diferentes para transações locais e distribuídas. As transações locais são processadas por um protocolo rápido. Neste protocolo, o *Walter* faz a verificação descrita acima, persiste as modificações e propaga as atualizações para as réplicas. As transações locais não precisam adquirir bloqueio, ao contrário das transações distribuídas. As transações distribuídas são processadas por um protocolo lento. Neste protocolo, o *Walter* utiliza um protocolo baseado no *two-phase commit* entre os servidores preferenciais para evitar conflitos com os outros processos que estão persistindo dados. Na primeira fase os bloqueios nos dados são adquiridos. Na segunda fase o servidor que está executando a transação persiste as modificações utilizando o protocolo rápido.

3.6 Sumário

Este capítulo apresentou sistemas de banco de dados NoSQL que utilizam mecanismos para controle de concorrência que são semelhantes ao descrito nesta proposta.

Todos os sistemas, exceto o *Walter* [Sovran et al., 2011], garantem a serialização das transações. Os autores admitem um grau de consistência mais fraco pois, ele não causa impacto no ambiente caracterizado para a utilização do sistema.

Em relação aos mecanismos para controle de concorrência, as abordagens propostas por [Padhye e Tripathi, 2015] e [Du et al., 2013] utilizam apenas o isolamento por *snapshot*. As abordagens propostas em [Padhye et al., 2014] e [Sovran et al., 2011] utilizam mecanismos baseados no isolamento por *snapshot*, e propõem uma extensão para dar suporte à consistência

causal. O *Spanner* [Corbett et al., 2013] utiliza mecanismos distintos de acordo com tipo de transação. É utilizado um mecanismo baseado em *two-phase locking* para as atualizações, e o isolamento por *snapshot* para transações somente de leitura.

No que diz respeito à aquisição de bloqueios nos protocolos baseados em *timestamp ordering*, o único mecanismo que não requer bloqueio é o *Clock-SI*. O mecanismo proposto por [Padhye e Tripathi, 2015] requer bloqueios para todas as operações para evitar anomalias na serialização. Os mecanismos propostos em [Padhye et al., 2014] e [Sovran et al., 2011] necessitam de bloqueio apenas para as atualizações.

No que diz respeito à utilização de cache, o *Walter* [Sovran et al., 2011] faz cache dos dados mais acessados. O *Spanner* [Corbett et al., 2013] mantém todas os dados modificados em *buffer*, e periodicamente faz a persistência em disco. Nas operações de leitura, o mecanismo do *Spanner* [Corbett et al., 2013] cria uma visão com os dados do *buffer* e do disco.

	Padhye et al.	PCSI	Clock-SI	Spanner	Walter	Alocs
mecanismo	IS	IS	IS	2V2PL	IS	IS
protocolo	TO	TO	TO	2PL / TO	TO	TO
bloqueio s. leitura	sim	não	não	não	não	não
resolução conflitos	lock	lock	TO	lock	lock	lock
consistência	centralizado	vet. relógios	rel. físicos	rel. físicos	vet. relógios	vet. relógios
serialização	sim	sim	sim	sim	sim	sim
cache	não	não	não	não	s. leitura	sim

Figura 3.1: Comparativo dos Trabalhos Relacionados

Esta dissertação trata sobre a adição de um módulo para controle de concorrência no ALOCS [Bungama et al., 2016], propondo um modelo para gerenciamento de transações que utiliza o isolamento por *snapshot* como mecanismo para o controle de concorrência com o protocolo *timestamp ordering*. O ALOCS utiliza cache para as operações de leitura e escrita, agrupa os pares chave-valor em *buckets* que são unidades de localização e transferência de dados.

Os modelos propostos por [Padhye et al., 2014], [Padhye e Tripathi, 2015] e [Du et al., 2013] utilizam o mesmo mecanismo que o proposto para o ALOCS, mas não utilizam cache para o processamento das operações. A opção por não utilizar o cache obriga os sistemas baseados nestes modelos a fazerem vários acessos remotos para a obtenção dos dados requeridos. O modelo proposto para o ALOCS reduzirá os acessos remotos durante o processamento das operações pois, os pares chave-valor são organizados em *buckets*. Os *buckets* funcionam como unidade básica para transmissão de dados. Ao requisitar um *bucket* remoto, este será mantido localmente no cache.

O modelo proposto por [Sovran et al., 2011] implementado no banco de dados *Walter* é semelhante ao ALOCS. Ele mantém os dados em objetos, e cada objeto tem um par chave-valor associado. Os objetos que tenham correlação são agrupados em *contêineres*. Apesar de agrupar os pares em *contêineres*, quando um dado não disponível localmente é solicitado, é feito somente a transferência do objeto relacionado. O Walter mantém um histórico dos pares lidos que funciona como cache somente-leitura.

O modelo proposto por [Corbett et al., 2013] é direcionado para replicação síncrona de dados. As leituras são direcionadas para o líder do grupo *Paxos* e não há cache.

Capítulo 4

ALOCS - Repositório chave-valor com controle de localidade de dados

Este capítulo descreve a arquitetura de um repositório chave-valor, denominado ALOCS [Bungama et al., 2016]. Este sistema permite a alocação de dados em sistemas de armazenamento distribuído, com controle de localidade. Este controle é obtido através do agrupamento de pares chave-valor em uma única estrutura, denominada *bucket*, cuja localidade física é controlada pela aplicação.

4.1 Modelo de Dados

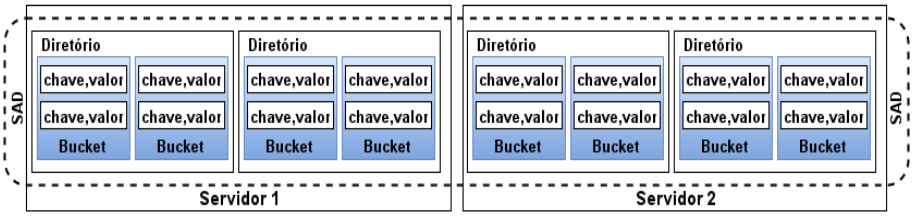


Figura 4.1: Representação do Modelo de Dados. [Bungama et al., 2016]

A Figura 4.1 é uma representação do modelo de dados. A unidade de armazenamento é um par baseado no modelo chave-valor. Os pares chave-valor são alocados em uma estrutura denominada *bucket*. Os *buckets* são agrupados em diretórios definidos como unidade de replicação. O servidor agrega um ou mais diretórios. Os identificadores dos diretórios são únicos para cada servidor, e os identificadores dos *buckets* são distintos dentro de cada diretório.

O *bucket* é uma estrutura física que agrupa os pares chave-valor gerenciados pela aplicação, utilizado como unidade de movimentação e localização física dos dados armazenados. O tamanho do *bucket* pode ser configurado de acordo com a necessidade da aplicação. Se em uma operação de escrita o limite de tamanho for extrapolado, ocorre uma operação de *split*, que cria um novo *bucket* para transferir as chaves excedentes.

O controle de localidade é baseado em intervalos de chaves associados aos *buckets*. O mapeamento entre intervalos e *buckets* é mantido por um sistema de metadados, que é consultado em todas as operações executadas pela aplicação. Quando a aplicação requisita uma chave, o *bucket* em que a chave foi alocada é identificado através do intervalo de chaves ao qual a chave

pertence. O ALOCS retorna para a aplicação apenas o par requisitado. Os dados para localização dos pares são mantidos no cabeçalho do *bucket*.

4.1.1 Metadados

Os metadados são informações sobre a localidade dos *buckets* armazenados no sistema de armazenamento utilizado pela aplicação. A localidade é expressa por meio de um caminho que reflete a hierarquia do modelo de dados. Todos os componentes do modelo, vistos na seção 4.1, recebem identificadores únicos. Tendo como exemplo um *bucket* identificado como *Habitantes*, em um diretório denominado *Municipio* e um Servidor identificado por *Estado*, e considerando que o *bucket* armazene chaves no intervalo de 1 a 500, a localidade do mesmo pode ser codificada da forma: ***[1-500] -> /Estado/Municipio/Habitantes***. Estas informações são coletadas durante a criação dos *buckets*.

4.2 Arquitetura

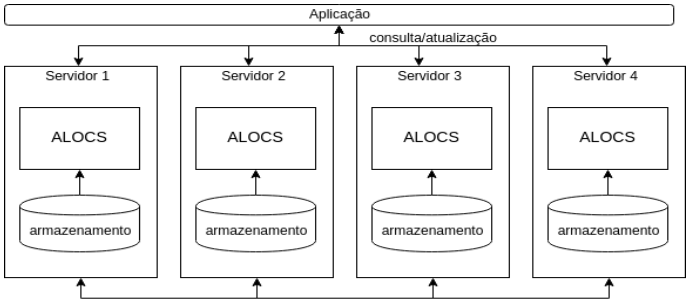


Figura 4.2: Arquitetura de Processamento.

O diagrama na Figura 4.2 ilustra a arquitetura de processamento proposta para utilização do repositório. Um ou mais servidores executam uma instância do ALOCS que é utilizado pela aplicação como infraestrutura de armazenamento. Cada Servidor contém uma base de dados local e um cache.

A aplicação pode direcionar as requisições do cliente para qualquer um dos servidores. As operações que envolverem dados que não fazem parte da base de dados local são realizadas através da transmissão de dados entre os servidores.

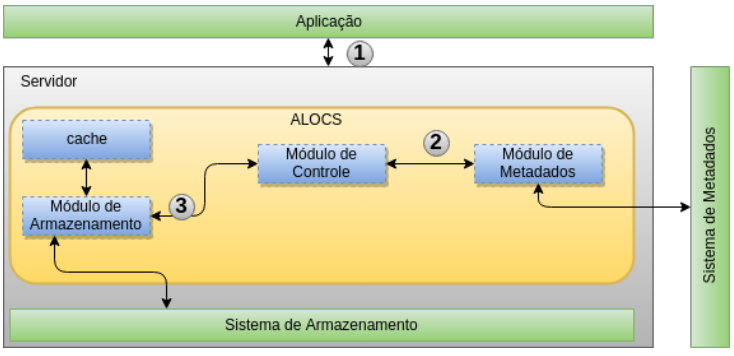


Figura 4.3: Arquitetura do ALOCS.

Os componentes que fazem parte da arquitetura do ALOCS estão ilustrados na Figura 4.3. A comunicação com os sistemas adjacentes, é realizada através de módulos por meio de

operações específicas. As interfaces dos módulos de armazenamento e metadados podem ser implementadas de acordo com os sistemas utilizados.

De maneira geral o fluxo de execução é: a aplicação requisita uma sequência de operações ao módulo de controle, que por sua vez solicita informações sobre a localidade dos dados ao módulo de metadados, e baseado na localidade fornecida, encaminha as operações solicitadas ao módulo de armazenamento.

O sistema de metadados mantém informações sobre as estruturas do modelo de dados criadas pela aplicação, e o mapeamento entre *buckets* e os intervalos de chaves. Estas informações são consultadas pelo módulo de controle para obter a localização dos *buckets* em que as chaves manipuladas pela aplicação foram alocadas.

4.2.1 Cache

O ALOCS mantém em cache os *buckets* acessados pela aplicação durante o processamento das consultas. Esta funcionalidade permite ao ALOCS retornar pares que estejam localizados no mesmo *bucket* sem fazer novas leituras em disco ou transmissões de dados, reduzindo o tempo de resposta das consultas que envolvam *buckets* acessados anteriormente.

O gerenciamento do cache é responsabilidade do Módulo de Armazenamento. O tamanho do cache pode ser configurado de acordo com a carga de trabalho da aplicação. Atualmente o ALOCS está configurado para manter 100 *buckets* com tamanho de 64K em cache.

Durante o processamento das operações o ALOCS faz acessos ao disco somente nas primeiras requisições aos *buckets*. O módulo de armazenamento é o responsável por requisitar os *buckets* ao sistema de armazenamento e copiá-los para o cache. Se não houver uma linha de cache disponível, o módulo utiliza uma política de renovação baseada no algoritmo LRU [Karedla et al., 1994].

4.2.2 Módulos

Esta seção descreve brevemente os módulos que fazem parte da arquitetura do ALOCS, e como ocorre a comunicação através das interfaces que os compõem.

4.2.2.1 Módulo de Controle

O módulo de controle é responsável por receber as requisições da aplicação, encaminhá-las para o módulo de armazenamento, e comunicar-se com o módulo de metadados para obter a localidade dos pares chave-valor solicitados pela aplicação.

Este módulo contém uma interface que faz a comunicação da aplicação com o ALOCS, através de operações que possibilitam a criação, remoção e busca dos componentes do modelo de dados descrito na seção 4.1. Nestas operações não é necessário conhecer o *bucket* em que os pares gerenciados foram alocados pois, esta informação é obtida através do módulo de metadados baseado no mapeamento entre os intervalos de chaves e os *buckets*.

A seguir é apresentada uma breve descrição das operações disponíveis na interface.

- ***clean()**:* Requisita a remoção do sistema de armazenamento de todos os *buckets* que estiverem vazios, ou com versões antigas. O retorno da operação é a confirmação de execução da operação.

- ***create_server(idServer)**:* Requisita a criação de um servidor, com o nome que foi especificado por parâmetro. O retorno da operação é a confirmação de execução da operação.

- ***create_dir(idDirectory,idServer):*** Requisita a criação de um Diretório, no Servidor que foi especificado nos parâmetros de entrada. O retorno da operação é a confirmação de execução da operação.

- ***replicate_dir(idDirectory,idServer):*** Requisita a replicação de um Diretório, especificado nos parâmetros de entrada em conjunto com o Servidor de destino. O retorno da operação é a confirmação de execução da operação.

- ***drop_dir(idDirectory,idServer):*** Requisita a remoção de um Diretório do Servidor especificado nos parâmetros de entrada. Se o Diretório não possuir réplicas, a remoção será permitida somente se o mesmo estiver vazio. O retorno da operação é a confirmação de execução da operação.

- ***dropALL_dir(idDirectory):*** Requisita a remoção do Diretório especificado como parâmetro de entrada. Nesta ação todas as réplicas do Diretório especificado também são removidas. O retorno da operação é a confirmação de execução da operação.

- ***create_bucket(idTx,idBucket,idDirectory,iniKey,finKey):*** Requisita a criação de um *bucket* no Diretório que foi especificado nos parâmetros de entrada em conjunto com o identificador do *bucket*, e o intervalo de chaves associado ao *bucket*. A criação do *bucket* é associada à transação identificada por *idTx*. O retorno da operação é a confirmação de execução da operação.

- ***drop_bucket(idTx,idBucket,idDirectory):*** Requisita a remoção do *bucket* no Diretório especificado nos parâmetros de entrada. A remoção do *bucket* está associada à transação identificada por *idTx*, e a condição para que seja realizada é do *bucket* estar vazio. O retorno da operação é a confirmação de execução da operação.

- ***put_pair(idTx,key,value):*** Requisita a adição de um par chave-valor com a chave e valor especificados nos parâmetros de entrada. A adição do par está vinculada à transação identificada por *idTx*. O retorno da operação é a confirmação de execução da operação.

- ***get_pair(idTx,key):*** Requisita um par chave-valor identificado pela chave especificada como parâmetro de entrada. A requisição do par está vinculada à transação identificada por *idTx*. O retorno da operação é o valor associado a *key*.

- ***remove_pair(idTx,key):*** Requisita a remoção de um par chave-valor identificado por uma chave especificada como parâmetro de entrada. A remoção do par está vinculada à transação identificada por *idTx*. O retorno da operação é a confirmação de execução da operação.

4.2.2.2 Módulo de Armazenamento

O módulo de armazenamento faz a comunicação entre o ALOCS e o sistema de armazenamento. É responsável por fazer o mapeamento entre os modelos de dados do ALOCS e sistema de armazenamento.

Este módulo contém uma interface que deve ser implementada de acordo com o sistema de armazenamento utilizado. Esta interface permite a comunicação entre o módulo de controle e o sistema de armazenamento, através de um conjunto de operações. O sistema de armazenamento deve ser compatível com o modelo de dados utilizado pelo ALOCS. É possível trocar o sistema de armazenamento, sem alterar a aplicação.

A seguir é apresentada uma breve descrição das operações disponíveis na interface.

- ***create_bucket(idServer,idDirectory,idBucket)**:* Cria um *bucket* em Servidor e Diretório específicos. Devem ser passados por parâmetro além do identificador do *bucket*, os identificadores do Servidor e do Diretório.

- ***drop_bucket(idServer,idDirectory,idBucket)**:* Remove um *bucket*, alocado em um Diretório e Servidor específicos. Devem ser especificados por parâmetro além do identificador do *bucket*, os identificadores do Diretório e Servidor. A condição para a execução da operação é o *bucket* estar vazio.

- ***create_dir(idDirectory,idServer)**:* Cria um Diretório em Servidor especificado por parâmetro.

- ***copy_dir(idDirectory,idServer1,idServer2)**:* Copia um Diretório de um Servidor para outro Servidor. Devem ser especificados por parâmetro além do identificador do *bucket*, os identificadores do Servidor de Origem, *idServer1*, e de Destino, *idServer2*.

- ***drop_dir(idDirectory,idServer)**:* Remove um Diretório de um Servidor. Deve ser especificado por parâmetro além do identificador do Diretório, o identificador do Servidor. A condição para a execução da operação é o Diretório estar vazio.

- ***get_pair(idServer,idDirectory,idBucket,key,ts)**:* Retorna o valor associado à chave *key* alocado no *bucket* localizado pelo Módulo de Metadados. Devem ser passados por parâmetro além do identificador do *bucket*, os identificadores do Servidor e Diretório, que são obtidos através dos metadados. Nesta operação o módulo de armazenamento coloca o *bucket* que será utilizado em uma linha de cache disponível, caso ainda não tenha sido acessado.

- ***put_pair(idServer,idDirectory,idBucket,key,value)**:* Adiciona um par chave-valor no *bucket* localizado pelo Módulo de Metadados. Devem ser especificados por parâmetro além do par a ser adicionado, os identificadores do Servidor, Diretório, e *bucket*, que são obtidos através dos metadados. Nesta operação o módulo de armazenamento coloca o *bucket* que será utilizado em uma linha de cache disponível, caso ainda não tenha sido acessado. Após a operação o *bucket* fica marcado como atualizado.

- ***remove_pair(idServer,idDirectory,idBucket,key)**:* Remove um par chave-valor no *bucket* localizado pelo Módulo de Metadados. Devem ser especificados por parâmetro além da chave a ser removida, os identificadores do Servidor, Diretório, e *bucket*, que são obtidos através dos metadados. Nesta operação o módulo de armazenamento coloca o *bucket* que será utilizado em uma linha de cache disponível, caso ainda não tenha sido acessado. Após a operação o *bucket* fica marcado como atualizado.

4.2.2.3 Módulo de Metadados

O módulo de metadados permite ao módulo de controle solicitar informações sobre os dados alocados pela aplicação ao sistema que controla os metadados. A comunicação é realizada através de um conjunto de operações disponíveis em uma interface.

A seguir é apresentada uma breve descrição das operações disponíveis na interface.

- ***put_server(idServer)**:* Adiciona aos metadados um servidor, criado pela operação *create_server*. O nome do servidor deve ser especificado.

- ***put_dir(idDirectory,idServer)**:* Adiciona aos metadados um Diretório criado pelas operações *create_dir* ou *copy_dir*. Além do Diretório, deve ser especificado o Servidor.
- ***get(idDirectory)**:* Retorna uma lista de Servidores, em que o Diretório especificado por parâmetro, deverão estar alocados.
- ***drop_dir(idDirectory,idServer)**:* Remove dos metadados um Diretório, que foi removido por um operação *drop_dir*. O Diretório e o Servidor devem ser especificados.
- ***put_bucket(idBucket,idDirectory,iniKey,finKey)**:* Adiciona aos metadados um *bucket* criado pela operação *create_bucket*. Além do identificador do *bucket* devem ser especificados, o Diretório e o intervalo de chaves.
- ***get(idBucket, idDirectory)**:* Retorna o Sevidor que o *bucket* especificado por parâmetro deve estar alocado. Caso o Diretório relacionado tenha sido replicado, a função retorna uma lista com todos os Servidores associados.
- ***drop_bucket(idBucket,idDirectory)**:* Remove dos metadados um *bucket*, que foi removido por um operação *drop_bucket*. O identificador do *bucket* e o Diretório devem ser especificados.
- ***get(key)**:* Retorna o Servidor, Diretório e *bucket* em que o par chave-valor, cuja chave foi especificada por parâmetro, deve estar alocado. Caso o Diretório relacionado tenha sido replicado, a função retorna uma lista com todos os Servidores associados.

4.2.2.4 Interação entre os Módulos através das Interfaces

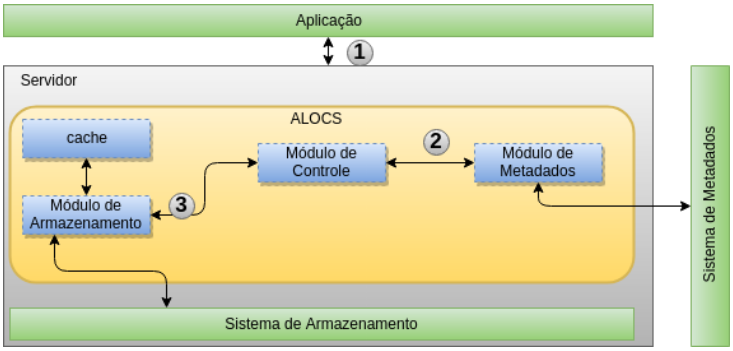


Figura 4.4: Passos para execução das operações.

O diagrama na Figura 4.4, ilustra o fluxo de operações entre os módulos. Para executar as consultas a aplicação comunica-se com o ALOCS, **passo 1**, através da operação ***get_pair(k)***. O módulo de controle recebe a requisição, e no **passo 2** solicita ao módulo de metadados a localização da chave requisitada, por meio da operação ***get(k)*** que retorna o Servidor, Diretório e *bucket*, baseado no intervalo de chaves. Após obter a localização das chaves, o módulo de controle solicita ao módulo de armazenamento o valor correspondente a chave no **passo 3**, por meio da operação ***get_pair(s,d,b,k)***. Se o *bucket* não estiver no cache, o módulo de armazenamento faz a solicitação para o sistema de armazenamento, se for da base de dados local, caso contrário solicita para o servidor de origem. Após recebê-lo o módulo de armazenamento o coloca em uma linha de cache disponível, e continua o processo para extração da chave solicitada pela aplicação.

Para executar as atualizações, a aplicação comunica-se com o ALOCS, **passo 1**, por meio da operação ***put_pair(k,v)***. O módulo de controle recebe a requisição, e no **passo 2** através da operação ***get(k)*** solicita ao módulo de metadados a localização do *bucket* em que a chave será alocada. Após obter a localização das chaves, no **passo 3** a atualização é encaminhada ao módulo de armazenamento através da operação ***put_pair(s,d,b,k,v)***. Após colocar o *bucket* em cache, o módulo de armazenamento executa a atualização solicitada.

4.3 Sumário

Este capítulo apresentou o ALOCS, um repositório chave-valor que permite a alocação de dados em sistemas de armazenamento distribuído, com controle de localidade. Este controle é obtido através do agrupamento de pares chave-valor em uma única estrutura, denominada *bucket*, cuja localidade física é controlada pela aplicação.

O modelo de dados do ALOCS permite agrupar pares chave-valor em *buckets*. O *bucket* é utilizado como unidade de armazenamento e transferência de dados.

O controle de localidade é obtido através de um sistema de metadados que mantém a localização dos *buckets*. O módulo de Controle faz a comunicação entre os sistemas de metadados e armazenamento.

O Alocs, como proposto por [Bungama et al., 2016] não contempla o gerenciamento de transações. A elaboração de um modelo para gerenciamento de transações e a implementação de um protocolo para controle de concorrência serão abordadas no próximo capítulo.

Capítulo 5

Um modelo de Gerenciamento de Transações para o ALOCS

5.1 Visão Geral

Nas aplicações para as quais o ALOCS foi projetado, as consultas e atualizações são executadas através de transações e podem envolver múltiplos *buckets*, não necessariamente alocados no mesmo servidor. Uma transação é constituída por uma sequência de operações de leitura e escrita delimitadas por *begin_tx* e *commit_tx*, ou *abort_tx* para encerrar, sendo executada de forma local e atômica através dos processos do ALOCS em conjunto com a aplicação.

Duas transações são consideradas concorrentes se estiverem ativas e acessando os mesmos *buckets*. Uma transação é considerada ativa enquanto estiver executando as operações de leitura e escrita.

O servidor que executa uma transação é denominado Servidor Executor, os servidores que contêm os *buckets* necessários para a execução da transação são denominados Servidores Origem.

5.2 Caracterização do ambiente

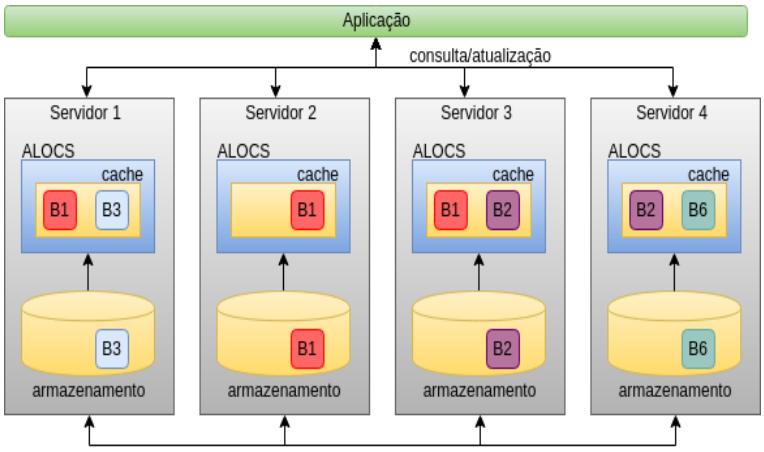


Figura 5.1: Caracterização do ambiente.

A Figura 5.1 ilustra a caracterização do ambiente que influenciou as decisões tomadas para a elaboração do modelo. Cada servidor mantém uma base de dados local e um cache. A aplicação pode direcionar as requisições do cliente para qualquer um dos servidores. As operações que envolverem dados que não fazem parte da base de dados local são realizadas através da transmissão de dados entre os servidores. O ALOCS recupera a localização dos dados através do módulo de metadados.

Na Figura 5.1 é possível observar o armazenamento dos *buckets* distribuídos entre os servidores, e um *snapshot* do conteúdo do cache durante o processamento de consultas. Destaca-se o fato dos servidores 1 e 3 estarem com o *bucket B1* em cache apesar do mesmo não estar alocado em suas bases de dados locais. Os servidores 1 e 3 podem atualizar o *bucket B1* localmente e enviar novas versões para serem persistidas no Servidor 2.

A motivação para a elaboração do modelo é manter a consistência das bases de dados locais após a atualização de *buckets* em situações semelhantes à ilustrada na Figura 5.1.

5.3 Controle de Concorrência Multiversão

O controle de concorrência multiversão baseado no isolamento por *snapshot* foi adotado como mecanismo de controle de concorrência para o ALOCS. A principal característica que determinou sua escolha é a imposição de um nível de isolamento reduzido possibilitando um alto grau de concorrência entre operações [Berenson et al., 1995], sem permitir a maioria das anomalias descritas na seção 2.2.1.

Para dar suporte ao modelo multiversão de dados, os *buckets* recebem versões que são incrementadas quando há alteração em seu conteúdo. A Figura 5.2 é uma representação deste modelo aplicado ao ALOCS. Todas as versões de um *bucket* são armazenadas no mesmo diretório da versão inicial.

Este modelo ainda não contempla a limpeza das versões não acessadas por um longo período de tempo, através, por exemplo, de um processo de *garbage collection*. Esta questão será tratada em trabalhos futuros.

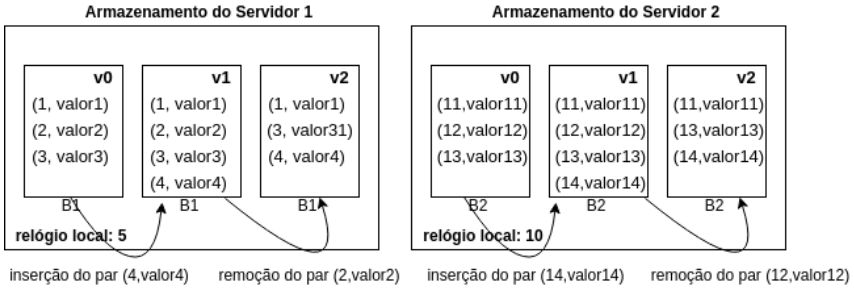


Figura 5.2: Representação do Modelo Multiversão de Dados.

5.3.1 Relógio Lógico Local

Como visto na seção 2.1.1 do capítulo 2, os relógios lógicos são utilizados para determinar a ordem em que um determinado evento aconteceu no sistema. Esta ordem, definida por uma sequência numérica, é importante para garantir a consistência na persistência dos dados.

No ALOCS cada servidor mantém um relógio lógico local utilizado para marcar o instante de tempo que ocorreu a última modificação na base de dados local. O relógio lógico é utilizado para delimitar a maior versão que pode ser lida de um *bucket*. Ele é incrementado na efetivação das modificações ocorridas nos *buckets* durante uma transação.

5.4 Componentes adicionados à arquitetura do ALOCS

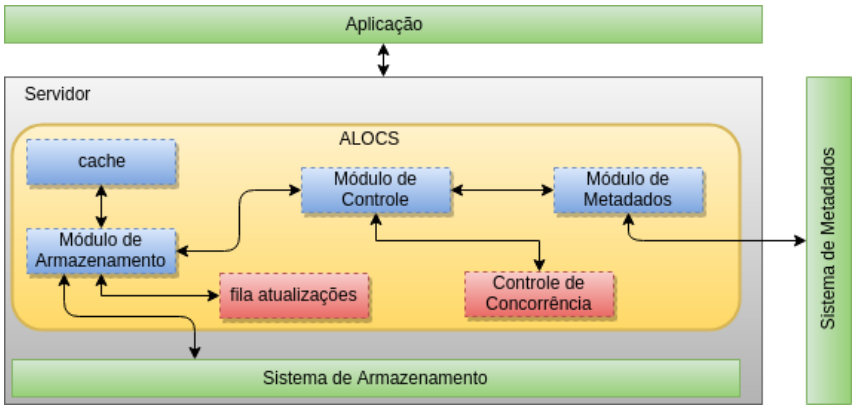


Figura 5.3: Componentes adicionados à arquitetura do ALOCS.

A Figura 5.3 ilustra a arquitetura do ALOCS com os componentes que foram adicionados ao ALOCS em destaque, e como eles interagem com os componentes já existentes.

5.4.1 Módulo de Controle

O Módulo de Controle da instância do ALOCS no Servidor Executor é o coordenador de execução das transações. Ele recebe as operações da aplicação e as encaminha para o Módulo de Armazenamento e para o Módulo de Controle de Concorrência.

Três operações foram adicionadas à interface do Módulo de Controle.

- ***begin_tx()***: Requisita o início de uma nova transação. O retorno da operação é o identificador da transação que foi criada.

- ***commit_tx(idTx):*** Requisita a efetivação da transação identificada pelo parâmetro *idTx*.
- ***abort_tx(idTx):*** Requisita a interrupção da transação identificada pelo parâmetro *idTx*.

Com relação as operações do ALOCS descritas na seção 4 as operações ***get_pair***, ***put_pair*** e ***remove_pair*** passaram a conter como parâmetros de entrada o *timestamp* de início da transação e o identificador da transação retornado pela operação *begin_tx*. A seguir estas operações são descritas com a adição destes parâmetros.

- ***get_pair(idTx,idServer,idDirectory,idBucket,key,ts)*** Retorna o valor associado à chave *key* alocado no *bucket* da versão determinada pelo *timestamp* de início da transação identificado por *ts*. Devem ser passados por parâmetro além do identificador do *bucket*, os identificadores do Servidor e Diretório, o *timestamp* de início da transação, e o identificador da Transação representado por *idTx*. O *timestamp* de início e o identificador da Transação são mantidos pelo Módulo de Controle de Concorrência.
- ***put_pair(idTx,idServer,idDirectory,idBucket,key,value,ts)*** Adiciona um par chave-valor no *bucket* da versão determinada pelo *timestamp* de início da transação identificado por *ts*. Devem ser especificados por parâmetro além do par a ser adicionado, os identificadores do Servidor, Diretório, e *bucket*, o *timestamp* do início da transação, e o identificador da Transação representado por *idTx*. O *timestamp* de início da transação e o identificador da Transação são mantidos pelo Módulo de Controle de Concorrência.
- ***remove_pair(idTx,idServer,idDirectory,idBucket,key,ts)*** Remove um par chave-valor no *bucket* da versão determinada pelo *timestamp* de início da transação identificado por *ts*. Devem ser especificados por parâmetro além da chave a ser removida, os identificadores do Servidor, Diretório, e *bucket*, o *timestamp* do início da transação, e o identificador da Transação representado por *idTx*. O *timestamp* de início da transação e o identificador da Transação são mantidos pelo Módulo de Controle de Concorrência.

5.4.2 Módulo de Controle de Concorrência

O Módulo de Controle de Concorrência é responsável por iniciar as transações, gerenciar os relógios lógicos, e coordenar a efetivação das transações.

O Módulo de Controle de Concorrência contém uma interface com três operações.

- ***begin_tx():*** Prepara o ALOCS para o início de uma nova transação. O retorno da operação é o identificador da transação que foi criada.
- ***commit_tx(idTx):*** Executa os processos para a efetivação da transação identificada pelo parâmetro *idTx*.
- ***abort_tx(idTx):*** O Módulo de Controle de Concorrência pode abortar uma transação que foi iniciada. Ao abortar uma transação o Módulo de Controle de Concorrência, retira do cache todos os *buckets* alterados durante a transação.

5.4.3 Módulo de Armazenamento

O Módulo de Armazenamento executa as operações no cache e faz a persistência dos dados na base de dados local. Ele controla a política de renovação no cache e a fila de atualizações.

5.4.3.1 Cache

Os *buckets* adicionados ao cache não são compartilhados entre as transações, apenas entre as operações associadas à transação. Esta decisão foi tomada para garantir que cada transação obtenha a versão do *bucket* referente ao *timestamp* do início da transação. Assim, é possível que diferentes transações executadas no mesmo servidor acessem versões distintas de um mesmo *bucket*.

Quando um *bucket* alterado por uma transação não efetivada é retirado do cache pela política de renovação, ele é inserido na fila de atualizações do servidor executor. Se durante a execução da transação o *bucket* retirado for requisitado novamente, o Módulo de Armazenamento pode recuperá-lo a partir da fila de atualizações.

5.4.3.2 Fila de Atualizações

Cada servidor contém uma fila de atualizações, gerenciada pelo Módulo de Armazenamento, para manter os *buckets* que foram modificados pelas transações. Nesta fila são mantidos duas categorias de *buckets*: *buckets* armazenados localmente e que tiveram dados alterados por transações efetivadas; e *buckets* alterados por transações não efetivadas, não necessariamente armazenados localmente, mas que foram retirados do cache pela política de renovação.

5.5 Protocolo para Controle de Concorrência

O protocolo para controle de concorrência implementado no ALOCS é baseado no isolamento por *snapshot*.

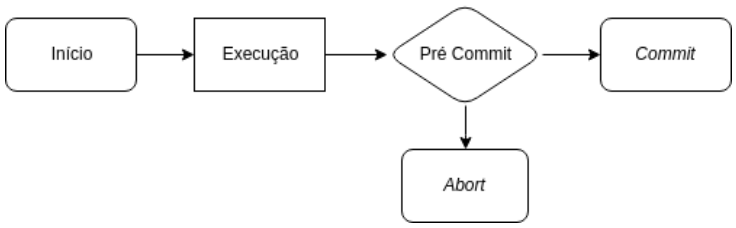


Figura 5.4: Fases do protocolo para controle de concorrência.

O protocolo para controle de concorrência passa por quatro fases, conforme ilustrado no diagrama da Figura 5.4. A fase Início ocorre quando o ALOCS recebe da aplicação a operação ***begin_tx***. A fase Execução dura enquanto o ALOCS receber da aplicação as operações ***get_pair***, ***put_pair*** e ***remove_pair*** da aplicação. A fase Pré-*Commit* é iniciada após o ALOCS receber da aplicação a requisição para encerrar a transação.

Na fase de Início o Módulo de Controle de Concorrência solicita os relógios locais de todos os servidores que estejam executando o ALOCS, e cria o relógio lógico global para guiar o Servidor Executor nas solicitações dos *buckets*.

O relógio lógico global é um vetor de relógios utilizado pelo Servidor Executor para especificar a versão dos *buckets* que devem ser enviados por um Servidor Origem.

Na fase Execução o Módulo de Armazenamento obtém os *buckets* necessários para a execução das operações. A primeira busca é feita no cache, caso o *bucket* não seja encontrado a fila de atualizações é verificada. Se houver uma atualização não persistida na fila, a mesma é removida da fila e inserida no cache. O Sistema de Armazenamento é acionado para leitura em disco somente em situações nas quais o *bucket* não foi encontrado nas buscas anteriores. Se o

bucket não estiver armazenado localmente, deve ser solicitado ao Servidor Origem. O Servidor Executor envia para o Servidor Origem o relógio lógico local, mantido pelo relógio lógico global, que recebeu na fase de Início.

Quando um Servidor Origem recebe uma solicitação por um *bucket* o Módulo de Armazenamento verifica primeiro a fila de atualizações. Caso o *bucket* seja encontrado na fila, ele é persistido. O Servidor Origem envia a maior versão que seja menor igual ao relógio local enviado pelo Servidor Executor. Se o *bucket* não for encontrado na fila, o Servidor Origem busca a versão requisitada na base de dados local.

Na fase Pré-Commit ocorre a validação das atualizações ocorridas na transação. A validação consiste em verificar se há conflitos entre as versões enviadas pelo Servidor Executor e as versões existentes nos Servidores Origem. A versão não deve ser menor ou igual nenhuma das versões dos *buckets* atualizados nos Servidores Origem.

A fase de validação é iniciada localmente pelo Servidor Executor caso tenha ocorrido modificações em sua base de dados. Se houver conflitos a fase Pré *Commit* é interrompida e o protocolo entra na fase *Abort*, na qual todas as modificações ocorridas na transação são descartadas e os *buckets* afetados são retirados do cache.

A partir do momento que o protocolo entra na fase de Pré-*Commit* é exigido um bloqueio de escrita na fila de atualizações dos servidores com bases de dados atualizadas. Este bloqueio é liberado somente após o encerramento da fase de *Commit*, ou com a interrupção da transação causada por uma operação ***abort_tx***. Durante o período do bloqueio, os servidores com as filas bloqueadas poderão fazer validações ou efetivar transações apenas para a transação que obteve o bloqueio.

Na fase de *Commit* as atualizações ocorridas durante a transação são registradas na fila e o relógio lógico local dos servidores onde houve modificação de dados é incrementado em 1. Esta fase ocorre somente após todos os servidores que receberam o pedido de validação retornarem ***accepted***, caso contrário o protocolo passa para a fase *Abort*.

O processo de registro é iniciado no Servidor Executor caso tenham ocorrido modificações em sua base de dados local. Após registrar as atualizações locais o Servidor Executor deve solicitar aos servidores que receberam o pedido de validação, o registro das atualizações que eles receberam na fase de Pré-*Commit*. O Servidor Executor deve aguardar o retorno dos servidores para enviar um pedido para encerrar a transação. Se um servidor retornar que houve falha no processo, o protocolo passa para a fase *Abort*. Após receber a confirmação de encerramento dos Servidores Origem, o Servidor Executor encerra a transação localmente.

A sequência de figuras a seguir ilustram um exemplo de execução do protocolo para controle de concorrência.

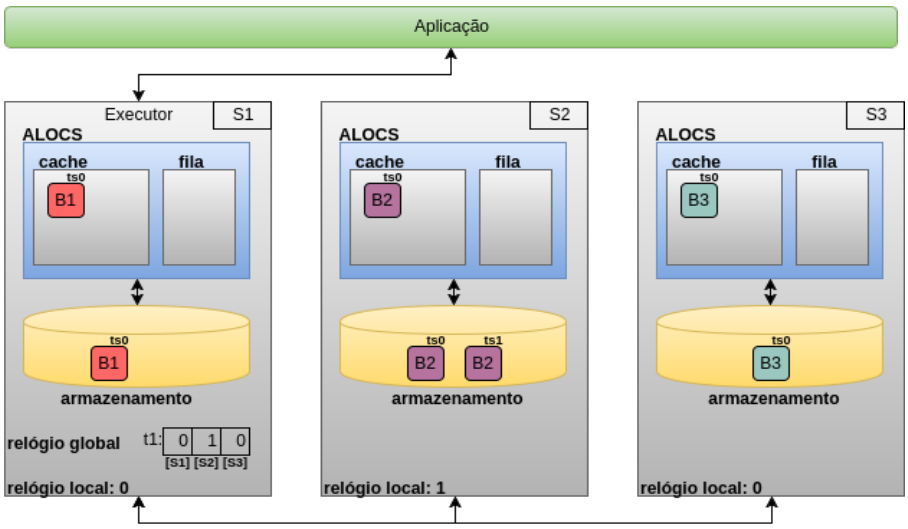


Figura 5.5: Fase de Início

A Figura 5.5 descreve a fase de Início do protocolo. O Servidor S1 recebe da aplicação a operação *begin_tx*, que é associada a um identificador (t_1) e solicita aos servidores S2 e S3 os relógios locais. O Módulo de Controle de Concorrência cria o relógio lógico global para a transação, que corresponde a um vetor de relógios locais. Assim no exemplo $t1.rGlobal[s_1] = 5$,$t1.rGlobal[s_2] = 10$ e $t1.rGlobal[s_3] = 5$.

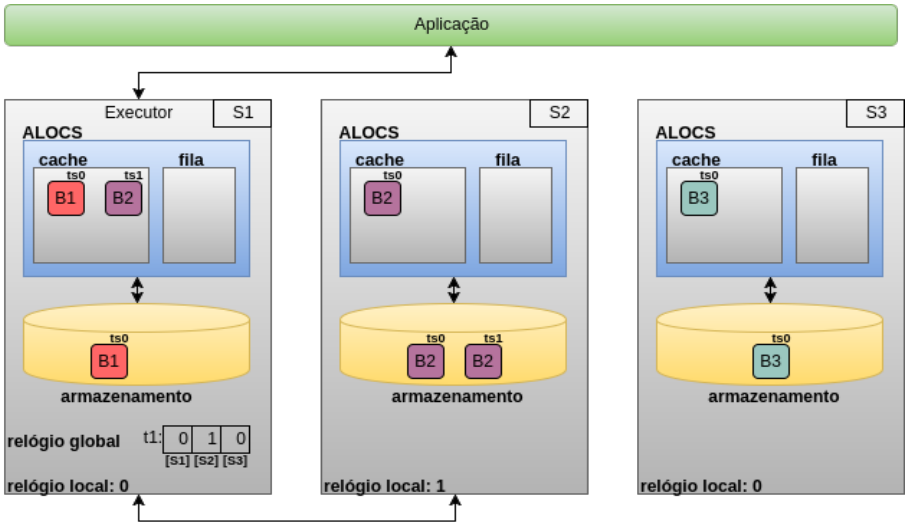


Figura 5.6: Fase de Execução - operação ***get_pair***

Após obter o relógio lógico global, o protocolo entra na fase de Execução onde o Servidor Executor recebe as operações ***get_pair***, ***put_pair*** e ***remove_pair*** enviadas pela aplicação.

A Figura 5.6 ilustra a fase Execução do protocolo para a operação ***get_pair***. O servidor S1 recebe da aplicação a operação ***get_pair(t1,2)***. O Módulo de Controle obtém a localidade da chave 2 do servidor de metadados, que é o *bucket* **B2** no servidor S2, e passa para o Módulo de Armazenamento. Conforme ilustrado na Figura 5.5 o *bucket* **B2** não está no cache de S1. Portanto o Módulo de Armazenamento deve requisitá-lo ao servidor S2 através da chamada

get_bucket(t1,S2,dir1,B2,10). Observe que o último parâmetro da chamada é o relógio local do servidor S2 no início da transação. Assim, a versão enviada por S2 deve ser menor ou igual **10**.

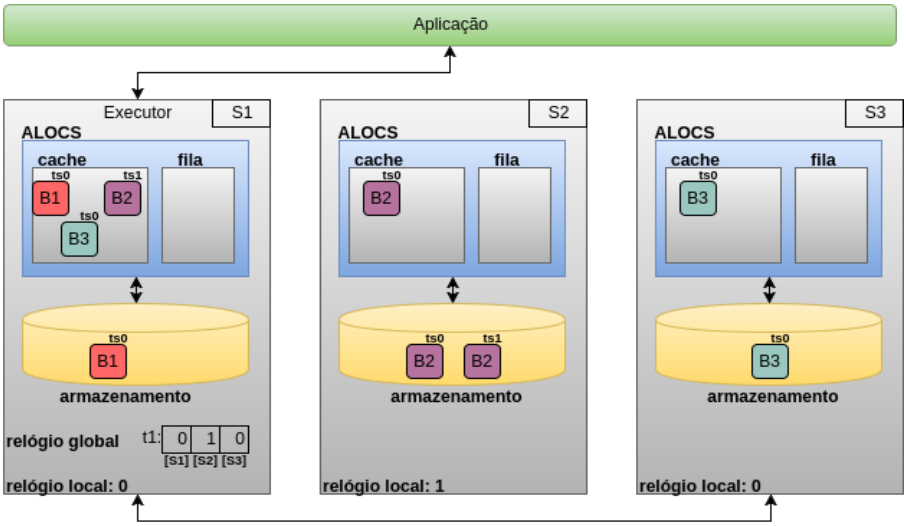


Figura 5.7: Fase de Execução - operação ***put_pair***

A Figura 5.7 ilustra a fase de Execução do protocolo para a operação ***put_pair***. O servidor S1 recebe da aplicação a operação ***put_pair(t1,10,valor)***. O Módulo de Controle obtém a localidade da chave 10, que é o *bucket* **B3** no servidor S3 e passa para o Módulo de Armazenamento. Conforme a Figura 5.5 o *bucket* **B3** não está no cache de S1, portanto o Módulo de Armazenamento deve requisitá-lo ao servidor S3. A versão enviada por S3 deve ser menor ou igual ao valor do relógio lógico enviado pelo servidor S1 que é **5**.

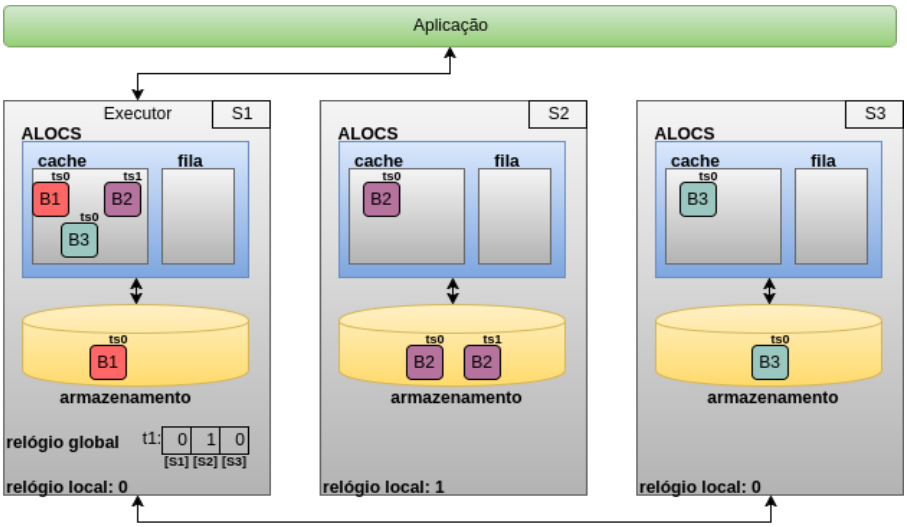


Figura 5.8: Fase de Pré-*Commit*

A Figura 5.8 ilustra a fase de Pré-*Commit* do protocolo iniciada após a aplicação enviar a operação ***commit_tx(t1)***. O servidor S1 envia através do Módulo de Controle de Concorrência uma solicitação de validação para o servidor S3. O servidor S3 faz o bloqueio em sua fila de atualizações e inicia a busca por conflitos entre a versão enviada por S1 e as versões existentes na fila e base de dados local. Se não houver conflitos entre as versões, S3 deve retornar para S1

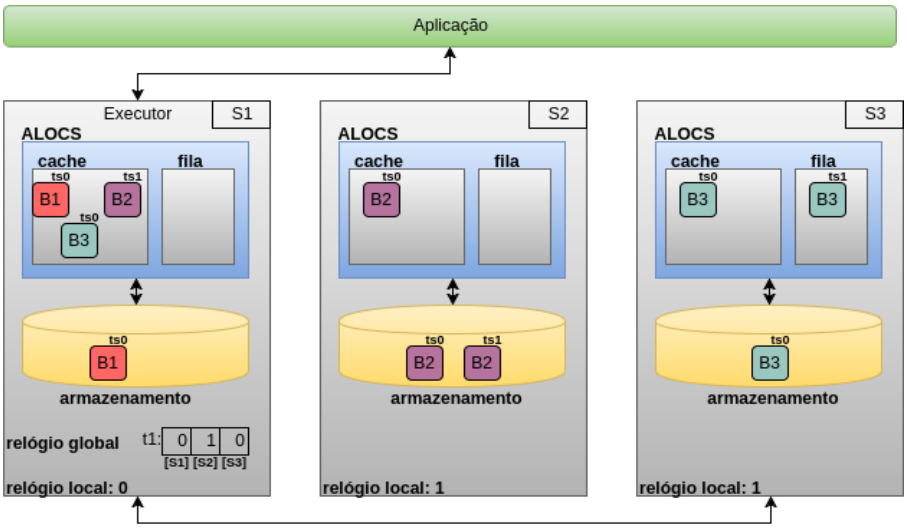


Figura 5.9: Fase de *Commit*

uma mensagem ***accepted***, caso contrário ***denied***. A partir deste momento até o encerramento da transação T_1, S3 poderá fazer validações e registrar atualizações somente para a transação T_1.

A Figura 5.9 descreve a fase de *Commit* do protocolo iniciada após o servidor S1 receber o retorno do servidor S3. O servidor S3 faz o registro das atualizações e envia o retorno para o servidor S1. A versão do *bucket* **B3** que está na fila, será efetivamente armazenada na base de dados somente no próxima vez que ele for requisitado por uma transação local ou remota.

Após receber o retorno do servidor S3, o servidor S1 envia um pedido para encerrar a transação. Ao receber o pedido de encerramento o servidor S3 deve retirar o bloqueio da sua fila de atualizações. Após a confirmação do encerramento da transação no servidor S3, o servidor S1 encerra a transação localmente.

5.6 Sumário

Este capítulo apresentou o modelo de gerenciamento de transações para o ALOCS. O controle de concorrência multiversão baseado no isolamento por *snapshot* foi adotado como mecanismo de controle de concorrência para o ALOCS. A consistência nas bases de dados locais é obtida através da utilização de relógios lógicos locais que marcam o instante de tempo em que houve a última modificação.

A motivação para a elaboração do modelo é manter a consistência das bases de dados locais após a efetivação das modificações ocorridas nos *buckets* durante a execução de transações locais ou remotas.

A partir do modelo foi elaborado um protocolo para controle de concorrência baseado no protocolo do isolamento por *snapshot*. O protocolo contém quatro fases: Início, Execução, Pré-*Commit* e *Commit*. O protocolo garante a consistência de dados mantidos em cache em um sistema de armazenamento distribuído. O próximo capítulo irá apresentar os resultados dos experimentos realizados para validar este protocolo.

Capítulo 6

Experimentos e Resultados

6.1 Implementação

Para implementação do protocolo a linguagem C foi utilizada em conjunto com a a API *ZeroMQ* [iMatix Corporation, 2014] para troca de mensagens.

Na etapa de implementação as operações ***get_pair***, ***put_pair***, ***remove_pair***, ***get_bucket*** e ***create_bucket*** já implementadas e descritas no capítulo 4 foram modificadas de acordo com o protocolo e modelo descritos no capítulo 5. O Apêndice A contém o detalhamento das estruturas e algoritmos implementados.

6.2 Ambiente

O ambiente definido para a realização dos experimentos contém 3 máquinas virtuais compostas por 2 núcleos intel core i5, 1GB de memória, 120GB de disco e sistema operacional Linux Fedora 25. Duas máquinas foram configuradas com o sistema de arquivos distribuído CEPH no modo *single node*, para executarem a aplicação com os experimentos, e a outra máquina com o Zookeeper para executar o módulo de metadados.

Em cada servidor de dados foi criada uma base de dados contendo um diretório e vinte e cinco *buckets* que armazenam 100 pares chave-valor. O valor associado a cada chave é aleatório e tem o tamanho máximo de 50 bytes. Estes valores foram definidos de forma experimental de acordo com a capacidade do hardware. Embora o CEPH permita a replicação de diretórios esta funcionalidade não foi utilizada na realização dos experimentos.

6.3 Experimentos e análise de resultados

Nesta seção serão apresentados três experimentos que tem como objetivo analisar o impacto da coalocação e da renovação do cache no algoritmo proposto neste trabalho. As métricas utilizadas são o tempo de resposta para a execução de todas as operações de uma transação e o *cache hit*.

$$cachehit = \frac{\sharp operacoes - \sharp acessos}{\sharp operacoes} \tag{6.1}$$

O *cache hit* é determinado pela equação 6.1. A variável **operações** é referente a quantidade de operações executadas na transação, a variável acessos é referente a quantidade de acessos ao disco.

$$coaloc = \sum \frac{\sharp paresAcessadosNoMesmobucket}{min(\sharp operacoes, \sharp paresTotalNobucket)} / (\sharp acessos * 100) \quad (6.2)$$

A coalocação pode ser descrita por meio da equação 6.2 sendo a relação entre o número de *buckets* acessados e a quantidade mínima de acessos necessários com os pares localizados nos mesmos *buckets*.

6.3.1 Experimento 1: impacto da coalocação na execução de consultas

O primeiro experimento teve como objetivo avaliar o impacto da coalocação das chaves na execução das consultas.

	% coalocação	pares por bucket	num. buckets
’ Cenário 1	2	1	50
Cenário 2	4	2	25
Cenário 3	10	5	10
Cenário 4	20	10	5
Cenário 5	100	50	1

Tabela 6.1: Definição das cargas de trabalho para o primeiro experimento

A tabela 6.1 mostra a definição das cargas de trabalho para o primeiro experimento. Cinco cenários foram criados para aplicar percentuais de coalocação pré-definidos. Além do percentual de coalocação, quatro parâmetros fixos foram utilizados: transações com 50 operações de leitura, cache de tamanho 1 e base de dados contendo 50 *buckets*, cada um contendo 100 pares chave-valor. A opção por manter o tamanho do cache em 1 foi para simular uma situação onde o cache não seria utilizado.

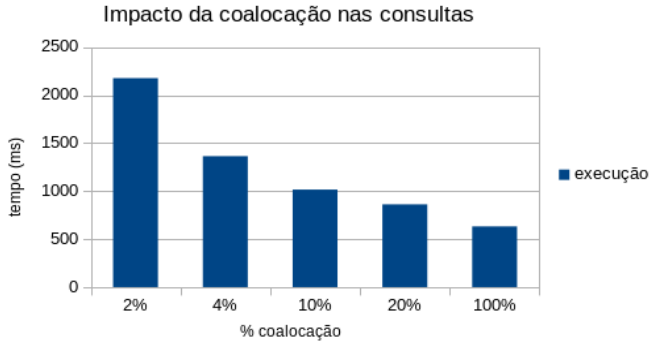


Figura 6.1: Resultados do Experimento 1

A Figura 6.1 contém o gráfico com os resultados. O tempo obtido para 1% de coalocação foi de 2176,692ms e para 100% de coalocação foi de 632,078ms, dando uma diferença de 1544,614ms. É importante observar que os pares foram lidos na sequência, então o fato do tamanho do cache estar configurado em 1, não interferiu no resultado.

Além disso, com 1% de coalocação houve comunicação entre dois servidores para envio dos *buckets*, pois, foram lidos 25 *buckets* locais e 25 *buckets* remotos. Este fator contribui para o

aumento do tempo de execução. Este experimento não avaliou o tempo de comunicação entre os servidores. Esta análise será feita em trabalhos futuros.

Através deste resultado é possível concluir que a coalocação adequada dos pares contribui para obter melhores resultados no processamento das consultas.

6.3.2 Experimento 2: efeito do cache na execução das consultas

O segundo experimento teve como objetivo avaliar o efeito do cache no processamento das consultas. Quatro cenários foram criados para aplicar tamanhos de cache pré-definidos. Além do tamanho do cache, três parâmetros fixos foram utilizados: transações com 50 operações de leitura, com uma base de dados de 50 *buckets*, cada um contendo 100 pares chave-valor. Para este experimento foi utilizado a mesma carga de trabalho para os 4 cenários: transações que executam leitura acessando 10 *buckets*.

Esta carga de trabalho é a mesma do Cenário 3 do Experimento 1. Contudo, no primeiro experimento as operações que acessavam o mesmo *bucket* eram executados em sequência. Para o novo experimento, a ordem das operações foi randomizada a fim de provocar renovação do cache.

Os experimentos foram executados com tamanhos de cache variando de 1 a 10.

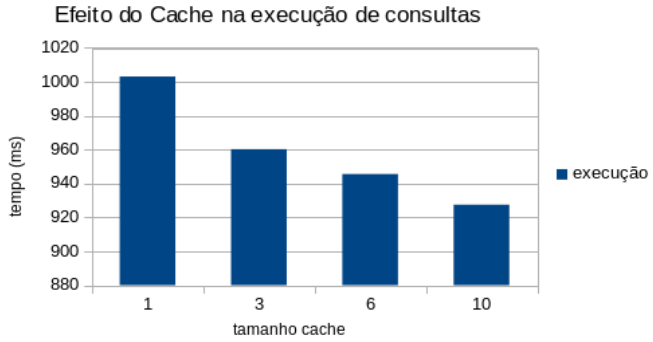


Figura 6.2: Resultados do Experimento 2

A Figura 6.2 contém o gráfico com os resultados. O tempo obtido com o cache de tamanho 1 foi de 1003,122ms com um *cache hit* de 0.56 e com o cache de tamanho 10 o tempo foi de 927,493ms com um *cache hit* de 0.8. Através deste resultado é possível observar que houve uma redução de 75,629ms do primeiro para o último cenário.

6.3.3 Experimento 3: impacto do controle de versão

O terceiro experimento teve como objetivo avaliar o impacto do controle de versão no processamento das atualizações. Para este experimento foram criados 5 cenários semelhantes ao primeiro experimento, o tipo de operação foi alterado para escrita e o tamanho do cache ajustado para 50, com o intuito da reposição do cache não interferir no resultado. As métricas utilizadas foram, tempo de execução e o tempo de *commit* para avaliar o quanto do tempo de execução foi gasto com a efetivação das atualizações.

A tabela 6.2 mostra a definição das cargas de trabalho para o terceiro experimento. Cinco cenários foram criados para aplicar percentuais de coalocação pré-definidos. Além do percentual de coalocação, quatro parâmetros fixos foram utilizados: transações com 50 operações

	% coalocação	pares por bucket	num. buckets
Cenário 1	2	1	50
Cenário 2	4	2	25
Cenário 3	10	5	10
Cenário 4	20	10	5
Cenário 5	100	50	1

Tabela 6.2: Definição das cargas de trabalho para o terceiro experimento

de escrita, cache de tamanho 1 e base de dados contendo 50 *buckets*, cada um contendo 100 pares chave-valor.

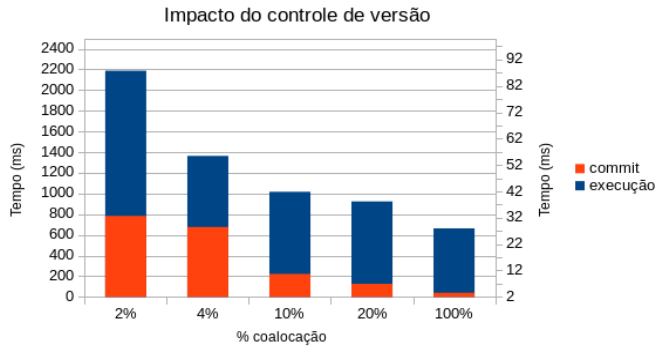


Figura 6.3: Resultados do Experimento 3

A figura 6.3 contém o gráfico com os resultados. O tempo de execução obtido no Cenário 1 foi de 2187,886ms sendo que 32.716ms foram utilizados para o *commit*. O tempo de execução obtido no Cenário 5 foi de 662,729ms sendo que 3.576ms foram utilizados para o *commit*. A redução do tempo de execução do primeiro cenário para o quinto foi de 1525,157ms.

Este resultado reforça a conclusão do Experimento 1, sobre a queda no tempo de execução das operações quando o percentual de coalocação aumenta. No Cenário 1 que forçou a comunicação entre dois servidores para o registro das atualizações houve um aumento de 29,14ms no tempo necessário para a efetivação das atualizações.

Em relação à fila de atualizações neste momento ele foi implementado apenas em memória para isolar o custo de gravação em disco, do custo do protocolo de controle de versão. É importante observar também que neste experimento não foi considerada a concorrência entre transações. Esta questão que ficará para trabalhos futuros.

6.4 Sumário

Este capítulo apresentou os experimentos elaborados com o intuíto de validar o protocolo para controle de concorrência do ALOCS. Os experimentos avaliaram o efeito da coalocação dos pares chave-valor nas operações de escrita e leitura, e o efeito do cache na execução das consultas. Os resultados dos experimentos relacionados à coalocação demonstraram a importância do controle de localidade para redução do tempo de execução das consultas. O uso do cache foi determinante para reduzir o tempo de execução das consultas.

Capítulo 7

Conclusão

O objetivo principal deste trabalho foi desenvolver uma solução para manter a consistência dos dados mantidos em cache e o sistema de armazenamento gerenciados pelo ALOCS. O ALOCS permite a alocação de dados em sistemas de armazenamento distribuído, com controle de localidade. Este controle é obtido através do agrupamento de pares chave-valor em uma única estrutura, denominada *bucket*, cuja localidade física é controlada pela aplicação.

As contribuições deste trabalho são a proposta de um modelo de armazenamento multiversão para o ALOCS, desenvolvimento de um protocolo para gerenciamento de transações que garante a consistência dos dados mantidos em cache em um sistema de armazenamento distribuído, a implementação do modelo proposto e um estudo experimental que mostrou o impacto da alocação dos dados sobre o desempenho do sistema, bem como o *overhead* do protocolo de controle de concorrência sobre o tempo de recuperação e escrita dos dados.

O controle de concorrência multiversão baseado no isolamento por *snapshot* foi adotado como mecanismo de controle de concorrência e permitiu ao ALOCS a capacidade de disponibilizar versões distintas de um mesmo *bucket* para transações executadas localmente ou remotas. A consistência nas bases de dados locais é obtida através da utilização de relógios lógicos locais que marcam o instante de tempo que houve a última modificação de dados. O protocolo para controle de concorrência é baseado no protocolo do isolamento por *snapshot* e contém quatro fases: Início, Execução, Pré-*Commit* e *Commit*.

Os experimentos realizados tiveram o objetivo de avaliar o efeito da coalocação dos pares chave-valor nas operações de escrita e leitura e o efeito do cache na execução das consultas. Os resultados dos experimentos relacionados à coalocação demonstraram a importância do controle de localidade para redução do tempo de execução das consultas. O uso do cache foi determinante para reduzir o tempo de execução das consultas.

Como trabalhos futuros pretende-se: criar um processo de *garbage collection* para limpeza das versões que não mais necessárias; implementar a fila de atualizações em disco; e realizar experimentos com um volume maior de dados, transações e servidores para determinar o custo de comunicação e avaliar questões de concorrência.

Referências Bibliográficas

[Agrawal et al., 2010] Agrawal, D., El Abbadi, A., Antony, S. e Das, S. (2010). Data management challenges in cloud computing infrastructures. *Databases in networked information systems*, páginas 1–10.

[ANSI, 1986] ANSI, X. (1986). American national standard for information systems: Database language sql. *American National Standards Institute, NY*, 135.

[Berenson et al., 1995] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. e O'Neil, P. (1995). A critique of ansi sql isolation levels. Em *ACM SIGMOD Record*, volume 24, páginas 1–10. ACM.

[Bernstein e Goodman, 1981] Bernstein, P. A. e Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221.

[Bernstein e Goodman, 1983] Bernstein, P. A. e Goodman, N. (1983). Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483.

[Bernstein et al., 1987] Bernstein, P. A., Hadzilacos, V. e Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison- Wesley.

[Bravo et al., 2015] Bravo, M., Diegues, N., Zeng, J., Romano, P. e Rodrigues, L. E. (2015). On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull.*, 38(1):18–31.

[Bungama et al., 2016] Bungama, P., Hara, C., de Oliveira, W. M. e Sousa, F. R. (2016). Um repositório chave-valor com controle de localidade. Em *SBBD*, páginas 88–99.

[Cattel, 2010] Cattel, R. (2010). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, páginas 12–27.

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. e Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[Cooper et al., 2008] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D. e Yerneni, R. (2008). PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1:1277–1288.

[Corbett et al., 2013] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P. et al. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. e Vogels, W. (2007). Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205.

[Du et al., 2013] Du, J., Elnikety, S. e Zwaenepoel, W. (2013). Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, páginas 173–184.

[DuBourdieux, 1982] DuBourdieux, D. (1982). Implementation of Distributed Transactions. Em *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, páginas 81–94.

[Fekete et al., 2005] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P. e Shasha, D. (2005). Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528.

[Gantz e Reinsel, 2012] Gantz, J. e Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16.

[iMatix Corporation, 2014] iMatix Corporation (2014). Zeromq distributed messaging. `http://zeromq.org/`. Acessado em 01/08/2017.

[Jacobs et al., 1995] Jacobs, K., Bamford, R., Doherty, G., Haas, K., Holt, M., Putzolu, F. e Quigley, B. (1995). Concurrency control, transaction isolation and serializability in sql92 and oracle7. *Oracle White Paper, Part*, (A33745).

[Karedla et al., 1994] Karedla, R., Love, J. S. e Wherry, B. G. (1994). Caching strategies to improve disk system performance. *Computer*, 27(3):38–46.

[Kumar et al., 2014] Kumar, K. A., Quamar, A., Deshpande, A. e Khuller, S. (2014). SWORD: workload-aware data placement and replica selection for cloud data management systems. *VLDB J.*, 23(6):845–870.

[Lakshman e Malik, 2010] Lakshman, A. e Malik, P. (2010). Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[Lamport, 1998] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169.

[Li et al., 2014] Li, Q., Wang, K., Wei, S., Han, X., Xu, L. e Gao, M. (2014). A data placement strategy based on clustering and consistent hashing algorithm in cloud computing. Em *Communications and Networking in China (CHINACOM), 2014 9th International Conference on*, páginas 478–483. IEEE.

[Mills, 1991] Mills, D. L. (1991). Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493.

[Padhye et al., 2014] Padhye, V., Rajappan, G. e Tripathi, A. (2014). Transaction management using causal snapshot isolation in partially replicated databases. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2014-Janua:105–114.

[Padhye e Tripathi, 2015] Padhye, V. e Tripathi, A. (2015). Scalable transaction management with snapshot isolation for NoSQL data storage systems. *IEEE Transactions on Services Computing*, 8(1):121–135.

[Paiva e Rodrigues, 2015] Paiva, J. e Rodrigues, L. (2015). On Data Placement in Distributed Systems. *ACM SIGOPS Operating Systems Review*, 49(1):126–130.

[Paiva et al., 2014] Paiva, J., Ruivo, P., Romano, P. e Rodrigues, L. (2014). Auto Placer. *ACM Transactions on Autonomous and Adaptive Systems*, 9(4):1–30.

[Shute et al., 2013] Shute, J., Vingralek, R., Samwel, B. e Rae, I. (2013). F1: A distributed SQL database that scales. Em *VLDB Endowment*, volume 6, páginas 1068–1079. ACM.

[Silva et al., 2015] Silva, J. A., Lourenço, J. M. e Paulino, H. (2015). Boosting locality in multi-version partial data replication. Em *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, páginas 1309–1314. ACM.

[Sovran et al., 2011] Sovran, Y., Power, R., Aguilera, M. K. e Li, J. (2011). Transactional storage for geo-replicated systems. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*, página 385.

[Tran, 2013] Tran, V.-T. (2013). *Scalable data-management systems for Big Data*. Tese de doutorado, École normale supérieure de Cachan-ENS Cachan.

[Vogels, 2009] Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1):40–44.

Apêndice A

Apêndice A

A.1 Estruturas de dados

Neste apêndice serão apresentados os algoritmos escritos para a implementação do protocolo para controle de concorrência.

A.1.1 Relógio lógico local

O relógio lógico local é mantido no Módulo Controle de Concorrência em uma variável denominada *l_clock*.

A.1.2 Relógio lógico global

O relógio lógico global é mantido no Módulo Controle de Concorrência em um vetor denominado *gclock_v*.

A.1.3 Registro *infotx_r*

O registro *infotx_r* é utilizado para armazenar os dados das transações em execução. O registro *infotx_r* é mantido pelo Módulo Controle de Concorrência e contém os seguintes atributos:

- ***integer idTx**:* identificador da transação;
- ***integer id_server**:* identificador do Servidor Executor;
- ***integer*[] ***gclock_v**:* relógio lógico global utilizado na transação identificada por *idTx*;
- ***update_r*[] ***updates**:* lista contendo os *buckets* modificados durante a transação.

A.1.4 Registro *update_r*

O registro *update_r* é utilizado para manter os dados referentes aos *buckets* atualizados durante a transação. O registro ***update_r*** é mantido pelo Módulo Controle de Concorrência e contém os seguintes atributos:

- ***local_r local**:* registro que armazena a localização, servidor/diretório/*bucket*, do *bucket* alterado;

- ***integer version**:* versão do *bucket* após a atualização. A versão do *bucket* é igual à versão do *bucket* lida durante a execução da transação, incrementada de 1 caso ele tenha sido atualizado;
- ***pointer data**:* ponteiro para uma linha do cache ou para o registro da fila, caso o *bucket* (alterado) tenha sido retirado do cache antes da transação ser encerrada.

A.1.5 Vetor *tx_list*

O vetor *tx_list* é mantido pelo Módulo Controle de Concorrência e utilizado para manter os dados das transações ativas. É composto por um conjunto de registros ***infotx_r***.

A.1.6 Registro *local_r*

O registro *local_r* armazena a localização do *bucket* obtida na operação ***get(key)***. O registro ***local_r*** contém os seguintes atributos:

- ***string id_server**:* identificador do servidor;
- ***string id_directory**:* identificador do diretório;
- ***string id_bucket**:* identificador do *bucket*.

A.1.7 Registro *line_r*

O registro *line_r* armazena informações sobre os *buckets* em cache. Ele contém os seguintes atributos.

- ***integer id_line**:* Identificador da linha do cache.
- ***local_r local**:* Localidade do *bucket*, composta por servidor/diretório/*bucket*, no Sistema de Armazenamento.
- ***integer version**:* Versão do *bucket* no Servidor Origem.
- ***integer dirty**:* Este atributo terá o valor verdadeiro quando houver uma alteração nos dados do cache.
- ***string data**:* Conteúdo do *bucket*.

A.1.8 Registro *queue_r*

O registro ***queue_r*** é utilizado para manter os *buckets* inseridos na fila de atualizações. Ele é utilizado como tipo de dados par o vetor *queue_v* mantido pelo Módulo de Armazenamento. O registro ***queue_r*** contém os atributos listados abaixo.

- ***integer idTx**:* Identificador da transação que escreveu os dados.
- ***integer tstamp**:* O tstamp tem valor -1 quando o *bucket* for inserido na fila pela política de renovação do cache. Quando o *bucket* é inserido na fila pelo processo de registro das atualizações tstamp recebe o valor do relógio local do Servidor Origem no momento do *commit*.

- ***local_r local:*** Localidade, servidor/diretório/*bucket*, do *bucket* que foi modificado.
- ***integer version:*** Versão do *bucket* no Servidor Executor se a transação foi efetivada, ou a versão do *bucket* no Servidor Origem se a transação não foi efetivada.
- ***integer id_server:*** Identificador do Servidor Executor.
- ***string data:*** Conteúdo atualizado do *bucket*.

A.2 Algoritmos

A.2.1 Execução das transações

Algoritmo 1: Algoritmo executado pelo Coordenador para execução das transações no Servidor Executor

```
1  queue_v ← {} ;                                                    /* queue with modified buckets */
2  tx_list ← {} ;                                                   /* list with active transactions */
3  while true do
4      case case op = begin(SE) do
5          idTx ← idTx + 1;
6          tx_list[idTx] ← begin_tx(idTx,SE);
7      case case op = get_pair(idTx,SE,key) do
8          gclock_v ← tx_list[idTx].infotx.gclock_v;
9          local ← get_location(key);
10         tstamp ← gclock_v[local.id_server];
11         value ← get_data(idTx,SE,tstamp,local,key);
12     case op = put_pair(idTx,SE,key,value) do
13         gclock_v ← tx_list[idTx].infotx.gclock_v;
14         local ← get_location(key);
15         tstamp ← gclock_v[local.id_server];
16         error ← put_data(idTx,SE,tstamp,local,key,value);
17     case op = remove_pair(idTx,SE,key) do
18         gclock_v ← tx_list[idTx].infotx.gclock_v;
19         local ← get_location(key);
20         tstamp ← gclock_v[local.id_server];
21         error ← remove_data(idTx,SE,tstamp,local,key);
22     case case op = commit(idTx,SE) do
23         infotx_r ← tx_list[idTx].infotx;
24         for all s in infotx_r.updates do
25             validation[SE] ← 'accept';
26         end
27         validation[SE] ← validate_updates(idTx,infotx_r.updates[SE]) ;          // prepare to commit
28         if validation[SE] = 'accept' then
29             for all s in (infotx_r.updates - SE) do
30                 send validate_updates(idTx,infotx_r.updates[s]) to s;
31             end
32             repeat
33                 response ← receive validation from s;
34                 validation[s] ← response;
35             until all array validation is filled up or timeout;
36             if all validation are 'accept' then
37                 register_updates(idTx,infotx_r.id_server,infotx_r.updates[SE]) ;          // commit
38                 for all s in (infotx_r.updates - SE) do
39                     send register_updates(idTx,infotx_r.id_server,infotx_r.updates[s]) to s;
40                 end
41                 repeat
42                     error ← receive response from s;
43                 until timeout or error = 1;
44                 if error = 0 then
45                     for all s in (infotx_r.updates - SE) do
46                         send end_commit(idTx,infotx_r.id_server) to s;
47                     end
48                 else
49                     abort(idTx,SE,infotx_r.updates[SE]);
50                     for all s in (infotx_r.updates - SE) do
51                         send abort(idTx,s,infotx_r.updates[s]) to s;
52                     end
53             else
54                 abort(idTx,SE,infotx_r.updates[SE]);
55                 for all s in (infotx_r.updates - SE) do
56                     send abort(idTx,s,infotx_r.updates[s]) to s;
57                 end
58         else
59             abort(idTx,SE,infotx_r.updates[SE]);
60 end
```

A.2.2 Algoritmo aplicado na operação *Begin*

Algoritmo 2: Algoritmo aplicado durante a execução da operação *begin* vista no Algoritmo 1

```
1  function begin_tx(idTx,SE) by SE
       Input: idTx → identificador único da transação, SE → identificador do Servidor Executor
       Output: infotx_r → registro com informações sobre transação
2      infotx_r.idTx ← idTx;
3      infotx_r.gclock_v ← get_global_clock(idTx,SE);
4      return infotx_r;

5  function get_global_clock(idTx,SE) by SE
       Input: idTx → identificador único da transação
       Output: gclock_v → vetor contendo o relógio lógico global
6      gclock_v[SE] ← l_clock;
7      for all s in(ALL_SERVERS - SE) do
8          send request_clock() to s;
9      end
10     repeat
11         server_clock ← receive local_clock_value(lc) from s;
12         gclock_v[s] ← server_clock;
13     until all array gclock_v is filled up or timeout;
14     return gclock_v;

15 function request_clock() by SO
16     send local_clock_value(l_clock) to SE;
```

A.2.3 Algoritmo aplicado ao Controle do Cache

Algoritmo 3: Algoritmo aplicado ao Controle do Cache

```
1  function get_bucket(idTx,tstamp,SE,local) by SE
       Input: idTx → id. da transação, tstamp → timestamp de início da transação para o SO,
              SE → id. do Servidor Executor,
              local → localização do Bucket,
              requires_locking → identifica operação que requer bloqueio - put_pair
       Output: bucket → ponteiro para o Bucket no Cache
2      data ← null;
3      bucket ← null;
4      line_r ← get_data_cache(idtx,tstamp,local.id_bucket);
5      if line_r = null then
           /* check_queue,verifica se o bucket foi retirado do cache pelo LRU antes
              de idTx ser encerrada, ou se há atualizações não persistidas na base
              de dados                                                              */
6          data ← check_queue(idTx,tstamp,local);
7          if data = null then
8              if local.id_server <> SE then
9                  send request_bucket(local,tstamp) to local.id_server;
10                 repeat
11                     data ← receive send_bucket(local,tstamp) from local.id_server;
12                 until data <> null or timeout;
13             else
14                 data ← read_data(local,tstamp);
15     if data <> null then
16         line_r ← put_data_cache(local,data,idTx,SE);
17         bucket ← line_r.data;
18     return bucket;
19 function check_queue(idTx,local,tstamp) by SE
       Input: idTx → id. da transação, tstamp → timestamp da transação,
              local → localidade do Bucket
       Output: bucket → conteúdo do Bucket
20     for all item in queue_v do
21         if item.local.id_bucket = local.id_bucket then
22             if item.tstamp < 0 and item.idTx = idTx then
                   /* bucket foi posto na fila pelo LRU antes do commit, e transação
                      requisitou novamente                                          */
23                 bucket ← item.data;
24             else if item.tstamp <= tstamp then
                   /* bucket foi modificado por transação encerrada                 */
25                 error ← persist_bucket(item.local,item.data,(item.version +1),item.tstamp);
26                 bucket ← item.data;
               /* retira o bucket da fila                                           */
27             dequeue(item);
28     end
29     return bucket;
```

Algoritmo 3: Continuação do algoritmo aplicado ao Controle do Cache

```
1  function persist_bucket(local,bucket,version,tstamp) by SE and SO
       Input: local → localidade do Bucket, bucket → conteúdo do Bucket,
              version → versão do Bucket no cache de SE, tstamp → tstamp de commit da transação
       Output: error → situação da operação
2      error ← write_bucket(local,version,tstamp,bucket);
3      return error;

4  function send_bucket(local,tstamp) by SO
       Input: local → localidade do Bucket, tstamp → timestamp de início da transação para o SO
       Output: data → conteúdo do bucket
5      data ← find_in_queue(local,tstamp);
6      if data = null then
7          data ← find_in_cache(local,tstamp);
8          if data = null then
9              data ← read_data(local,tstamp);
10     return data;

11 function get_data_cache(idTx,tstamp,local) by SE
       Input: idTx → identificador da transação tstamp → timestamp de início que define a versão do Bucket,
              id_bucket → identificador do Bucket
       Output: line_r → ponteiro para a linha de Cache
12     itcache_r ← null;
13     itcache_r ← search_data(idTx,tstamp,local);
14     if line_r <> null then
           /* move a linha de cache para o início da lista do LRU                    */
15         move_item_cache(itcache_r);
16     end
17     return line_r;

18 function put_data_cache(local,data,idTx,SE) by SE
       Input: local → localidade do bucket,
              data → bucket obtido do Servidor Origem,
              idTx → id. da transação que ira acessar o bucket SE → id. do Servidor Executor
       Output: line_r → ponteiro para a linha de cache
19     line_r ← get_line(SE);
20     if line_r = null then
21         itcache_r ← exec_policy();
22         if itcache_r <> null then
23             line_r ← itcache_r.line_r;
24             if line_r.dirty = 1 then
25                 error ← enq_data_cache(line);
26             end
27     if line_r <> null then
28         id_line ← id_line + 1;
29         line_r.id_line ← id_line;
30         line_r.local ← local;
31         line_r.version ← get_version(data) ;                  /* extrai versão do bucket */
32         line_r.data ← data;
33         line_r.id_tx ← idtx;
           /* adiciona o item de cache a lista do LRU                                */
34         if itcache_r <> null then
35             insert_itcache(line,it_cache);
36         else
37             atach_node(it_cache);
38         return line_r;
```

Algoritmo 3: Continuação do algoritmo aplicado ao Controle do Cache

```
1  function get_line(SE) by SE
       Input: SE → id. do Servidor Executor
       Output: line_r → ponteiro para o item de cache obtido
2      line_r ← null;
3      if check_limit() = 0 then
4          cache_indx ← cache_indx + 1;
5          line_r ← cache[cache_indx];
6      return line_r;
7  function exec_policy() by SE
       Output: itcache_r → item de cache menos acessado
8      itcache_r ← lrulist.last;
       /* retira o item de cache da lista do LRU                          */
9      detach_node(itcache_r);
10     return itcache_r;
11 function enq_data_cache(line_r;SE) by SE
       Input: line_r → registro contendo metadados do cache,
              SE → id. do Servidor Executor
       /* enq_data_cache coloca na fila apenas buckets de transações ativas */
12     enqueue(line_r.id_tx,-1,line_r.local,line_r.version,SE,line_r.data);
```

A.2.4 Algoritmo aplicado na execução de consultas

Algoritmo 4: Algoritmo aplicado no Módulo de Armazenamento para a execução de consultas

```
1  function get_data(idTx,SE,tstamp,local,key) by SE
       Input: idTx → id. da transação, SE → id. do Servidor Executor, tstamp → timestamp de início da transação para o
              SO,
              local → localização do Bucket em SO,
              key → chave que será consultada
       Output: value → valor correspondente a chave passada como parâmetro
2      bucket ← get_bucket(idTx,tstamp,SE,local);
3      value ← get_value(bucket,key,value);
4      return value;
```

A.2.5 Algoritmo aplicado na execução de atualizações

Algoritmo 5: Algoritmo aplicado no Módulo de Armazenamento para a execução de atualizações

```
1  function put_data(idTx,SE,tstamp,local,key,value) by SE
       Input: idTx → localização do Bucket em SO,
              SE → id. do Servidor Executor,
              tstamp → timestamp de início da transação para o SO local → localização do Bucket em SO,
              key → chave que será inserida,
              value → valor correspondente a chave
       Output: error → situação da operação
2      error ← 1;
3      bucket ← get_bucket(idTx,tstamp,SE,local);
4      if bucket <> null then
           /* após o par ser escrito, o vetor updates mantido por infotx é
              atualizado                                                  */
5          error ← write_pair(bucket,key,value);
6      return error;
```

A.2.6 Algoritmo aplicado na execução de remoções

Algoritmo 6: Algoritmo aplicado no Módulo de Armazenamento para a execução de remoções

```
1  function remove_data(idTx,tstamp,local,key) by SE
       Input: idTx → id. da transação, tstamp → timestamp de início da transação local → localização do Bucket em SO,
              key → chave que será consultada
       Output: error → situação da operação
2      error ← 1;
3      bucket ← get_bucket(idTx,tstamp,SE,local);
4      if bucket <> null then
           /* após o par ser removido, o vetor updates mantido por infotx é
              atualizado                                                    */
5          error ← remove(bucket,key);
6      return error;
```

A.2.7 Algoritmo aplicado na operação *commit*

Algoritmo 7: Algoritmo aplicado na operação *commit* descrita no Algoritmo 1

```
1  function check_conflicts(idTx,updates) by SE and SO
       Input: idTx → identificador da transação,
              updates → vetor contendo Buckets atualizados
       Output: valid → indica se a transação é valida
2      indx ← 0;
3      valid ← 1;
4      while updates[indx] <> null and valid = 1 do
5          id_bucket ← updates[indx].local.id_bucket;
6          for all q in queue_v do
7              if id_bucket = q.local.id_bucket then
8                  version ← q.version;
                   /* se a versão do bucket atualizado for <= a versão do bucket
                      encontrado na fila a transação não é valida               */
9                  if updates[indx].version <= version then
10                     valid ← 0;
11         end
12         indx ← indx + 1;
13     end
14     return valid;

15 function validate_updates(idTx,updates) by SE and SO
       Input: idTx → identificador da transação,
              updates → vetor contendo Buckets atualizados
       Output: validation → indica se a atualização foi aceita
16     lock on queue_v in updates.id_server;
17     if check_conflicts(idTx,updates) = 1 then
18         return 'accept';
19     else
20         return 'denial';

21 function register_updates(idTx,SE,updates) by SE and SO
       Input: idTx → identificador da transação,
              SE → id do servidor Executor,
              updates → vetor contendo Buckets atualizados de SE ou SO
22     l_clock ← l_clock + 1;
23     for all u in updates do
24         enqueue(idTx,l_clock,u.local,u.version,SE,u.data);
25     end
26     release lock on queue_v in updates[0].local.id_server;
```