UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DAVI BRAGA GOMES

ETERNAL: UMA ESTRATÉGIA EFICIENTE DE TOLERÂNCIA A FALHAS UTILIZANDO MEMÓRIA NÃO VOLÁTIL

FORTALEZA
2019

DAVI BRAGA GOMES

ETERNAL: UMA ESTRATÉGIA EFICIENTE DE TOLERÂNCIA A FALHAS UTILIZANDO MEMÓRIA NÃO VOLÁTIL

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Banco de Dados

Orientador: Prof. Dr. Javam de Castro Machado
Coorientador: Prof. Dr. Angelo Roncalli Alencar Brayner

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

G613e Gomes, Davi Braga.

ETERNAL: uma estratégia eficiente de tolerância a falhas utilizando memória não volátil. / Davi Braga Gomes. - 2019.

72 f : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciências da Computação, Fortaleza, 2019.

Orientador: Prof. Dr. Javam de Castro Machado.

Co-orientadora: Profª. Dr. Angelo Roncalli Alencar Brayner

1. Memória não volátil. 2. OLTP. 3. *Logging*. 4. Recuperação. I. Título.

CDD – 005

DAVI BRAGA GOMES

ETERNAL: UMA ESTRATÉGIA EFICIENTE DE TOLERÂNCIA A FALHAS UTILIZANDO MEMÓRIA NÃO VOLÁTIL

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Banco de Dados

Aprovada em: 29 / 11 / 2019

BANCA EXAMINADORA

Prof. Dr. Javam de Castro Machado (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Angelo Roncalli Alencar Brayner
Universidade Federal do Ceará (UFC)

Prof. Dr. José Maria da Silva Monteiro Filho
Universidade Federal do Ceará (UFC)

Prof. Dr. José de Aguiar Moraes Filho
Universidade de Fortaleza (UNIFOR)

À minha mãe Vanda e ao meu pai Airton, por sua inesgotável dedicação à nossa família.

AGRADECIMENTOS

Ao Prof. Dr. Javam Machado e à Profa. Dra. Rosélia Machado, que, por meio do Laboratório de Sistemas e Banco de Dados (LSBD), são responsáveis diretos pelo profissional, pelo pesquisador e pela pessoa que me tornei.

Ao Prof. Dr. Angelo Brayner, pelo seu total apoio durante a realização deste trabalho. Aos meus colegas do grupo de pesquisa de SGBD, pelas valiosas discussões realizadas durante as reuniões semanais. Em especial ao amigo Prof. Me. Paulo Amora, pelas diversas vezes que contribuiu com discussões além das reuniões.

Aos meu colegas do projeto Ldiag, que, em vários momentos, foram compreensivos com relação às vezes que precisei me ausentar do projeto devido ao mestrado. Em especial ao amigo Fernando Lima, que sempre se mostrou disposto a ajudar nos momentos de prazos apertados.

A todos os colegas do LSBD que, de uma forma ou de outra, me ajudaram durante a realização deste trabalho.

Aos meus pais, por nunca pouparem esforços quando se tratava de dar educação aos filhos.

A Lenovo e à CAPES, pelo financiamento do meu mestrado e desta pesquisa.

"*The game of science is, in principle, without end. He who decides one day that scientific statements do not call for any further test, and that they can be regarded as finally verified, retires from the game.*"

(Karl Popper)

RESUMO

SGBDs em memória principal têm se mostrado como alternativa eficiente para o gerenciamento de grandes volumes de dados. Eles caracterizam-se por utilizar memória RAM como seu meio de armazenamento primário. No entanto, tais sistemas precisam de armazenamento não-volátil para garantir a durabilidade de suas transações. Memórias não-voláteis endereçáveis por byte (NVRAM) são candidatas ideais para assegurar tal propriedade, pois possuem tempo de acesso próximo ao de memória RAM tradicional, mas ainda assim garantem a persistência dos dados. A fim de atacar o problema da durabilidade de transações em SGBDs em memória, este trabalho propõe ETERNAL, uma arquitetura de durabilidade que faz o uso eficiente de memória não- volátil, melhorando o desempenho do processo de persistência em tais SGBDs. Os experimentos revelam que ETERNAL provê um *throughput* superior a abordagem de escrita antecipada em log (WAL). Destaca-se ainda o fato, que, mesmo utilizando memória não-volátil como meio de armazenamento, a abordagem WAL não possui um mecanismo para lidar com o cenário em que a memória não volátil se esgota.

Palavras-chave: memória não volátil; OLTP; *logging*; recuperação.

ABSTRACT

In-memory DBMSs have proven to be an efficient alternative for managing large volumes of data. They're characterized by using RAM as their primary storage medium. However, such systems need nonvolatile storage to ensure the durability of its transactions. Byte addressable non-volatile memories (NVRAM) are ideal candidates for securing such a property because they have access times close to traditional RAM memory but still guarantee data persistence. In order to address the problem of transaction durability in main memory DBMSs, this paper proposes ETERNAL, a durability architecture that makes efficient use of nonvolatile memory, improving the process performance of persistence in such DBMSs. Experiments reveal that ETERNAL provides a higher throughput than the write-ahead logging (WAL) approach. It is also noteworthy that, even using nonvolatile memory as a storage medium, the WAL approach does not have a mechanism to deal with the scenario in which nonvolatile memory runs out.

Keywords: non-volatile memory; OLTP; logging; recovery.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

NVRAM	Memória Não Volátil de Acesso Aleatório
OLAP	Processamento Analítico em Tempo Real
OLTP	Processamento de Transações em Tempo Real
SSD	Unidade de Estado Sólido
WAL	*Write-ahead Logging*

SUMÁRIO

1 INTRODUÇÃO

O uso de sistemas de bancos de dados em memória principal (SBDMP) tem crescido significativamente. Esse fato ocorre principalmente em cenários cujas aplicações requerem alto desempenho, no que concerne a altas taxas de *throughput* e a baixo tempo de resposta de consultas. Mesmo com seus dados residindo primariamente em memória principal, um SBDMP ainda precisa garantir a persistência de deus dados em casos de perda do conteúdo da memória volátil, tais como erros de sistema ou quedas de energia. A maneira pela qual a persistência nesses sistemas é realizada são normalmente duas. A primeira é por meio de logs que registrem as modificações realizadas no banco, de tal maneira que seja possível reexecutar essas operações após uma queda do banco e recuperá-lo. A segunda maneira é por meio de *snapshots* periódicos que guardam todo o estado do banco em um meio de armazenamento persistente. Normalmente ambas as estratégias são utilizadas em conjunto.

Para garantir a durabilidade de transações confirmadas, a técnica mais utilizada em sistemas de banco de dados convencionais é a gravação antecipada de log (*write-ahead logging* – WAL) na qual qualquer transação só poderá ser confirmada após todas as suas alterações terem sido gravadas antecipadamente em um log. Tal técnica, no entanto, pode introduzir uma significante sobrecarga no desempenho de sistemas de banco de dados em memória (FAERBER *et al.*, 2017), pois, apesar das transações serem executadas rapidamente em memória principal, precisam necessariamente passar pela etapa de confirmação e escrita de registros no arquivo de log residente em memória não-volátil. Como se sabe, os dispositivos atuais de memória não-volátil, como discos rígidos (HDs) e unidades de estado sólido (SSDs), apresentam altos tempos de latência de acesso. Desta forma, para minimizar custo dessa etapa, um possível candidato para o armazenamento persistente do log é a nova classe de memórias não-voláteis endereçáveis por byte.

As memórias não-voláteis endereçáveis por byte (NVRAM) representam uma alter- nativa para incrementar significativamente o desempenho de sistemas que demandam persistência. Dentre as suas principais características, destacam-se a baixa latência de acesso aleatório, bem próxima de uma memória RAM convencional, e o baixo consumo energético, não sendo necessá- ria uma corrente elétrica permanente para garantir a persistência dos dados. Um dos principais impedimentos ao seu uso generalizado, no entanto, é seu maior custo quando comparado aos discos tradicionais, o que exige um uso eficiente. Uma comparação características das diversas tecnologias de armazenamento pode ser vista na Tabela 1 (ARULRAJ; PAVLO, 2017).

Tabela 1 – Comparação entre meios de armazenamento.

Tecnologia	Latência de Leitura	Latência de Escrita	Endereçabilidade	Volátil
HDD	10 ms	10 ms	Bloco	Não
SSD	25 μs	300 μs	Bloco	Não
NVRAM (PCM)	50 ns	150 ns	Byte	Não
RAM	60 ns	60 ns	Byte	Sim

Fonte: elaborada pelo autor.

Cargas de trabalho OLTP (Processamento de Transaçoes em Tempo Real) possuem um padrão de acesso *skewed* com relação à idade e ao uso do dado. Um *skew* com relação à idade significa que dados mais recentemente inseridos, e portanto mais novos, tendem a ser mais acessados que dados mais antigos. Um *skew* com relação ao uso significa que dados mais recentemente acessados possuem uma maior probabilidade de serem acessados posteriormente. O trabalho de (LEVANDOSKI *et al.*, 2013a) cita alguns exemplos: as cargas de trabalho de rastreamento de pacotes para empresas como a UPS ou a FedEx exibem *skew* de idade. Os registros de um novo pacote são atualizados com frequência até a entrega, depois são usados para análise por algum tempo e, depois disso, acessados novamente apenas em ocasiões raras. Outro exemplo citado é o desvio natural de sites de comércio eletrônico, como a Amazon, onde alguns itens são muito mais populares que outros. Tais preferências podem mudar o tempo, mas normalmente não muito rapidamente.

Em cargas de trabalho com *skew*, costuma-se chamar de quentes aqueles registros que possuem maior probabilidade de serem mais frequentemente acessados e de frios aqueles com menor frequência de acesso (ELDAWY *et al.*, 2014). Tipicamente, apenas alguns registros quentes são escritos mais frequentemente, porém, com o passar do tempo, ocorre diminuição da frequência de escrita e tais registros tornam-se frios, passando a ter menor probabilidade de serem escritos no futuro, eventualmente recebendo alguma leitura de alguma carga analítica. Em termos de desempenho, é interessante que tais registros quentes permaneçam em meios de armazenamento de acesso mais rápidos. Algumas abordagens que procuram diminuir o custo do meio armazenamento levando em conta o *skew* de cargas OLTP em SGBDs de memória principal já foram propostas anteriormente (DEBRABANT *et al.*, 2013) (AMORA *et al.*, 2018) (ZHANG *et al.*, 2016). Nenhuma, no entanto, tentou aproveitar essa característica no mecanismo de recuperação do sistema de banco de dados, o componente onde encontra-se o gargalo relacionado ao custo adicional da técnica WAL, para armazenamento de registros de log.

1.1 Contribuições

Este trabalho propõe ETERNAL, uma arquitetura de persistência para bancos de dados em memória principal que explora as propriedades de armazenamento híbrido para garantir a durabilidade de transações confirmadas. O problema que ETERNAL ataca é como minimizar a sobrecarga do sistema de recuperação em SBDMPs utilizando as capacidades únicas das memórias do tipo NVRAM em cargas de trabalho OLTP. A hipótese deste trabalho é de que memórias do tipo NVRAM permitem novas estratégias de persistência que apresentam desempenho superior às estratégias clássicas pensadas para disco que simplesmente inserem novas entradas em um log para cada atualização realizada no banco, principalmente em cargas de trabalho OLTP.

Diferente das estratégias clássicas baseadas em log, ETERNAL tira proveito da eficiência do acesso aleatório da NVRAM para persistir e manter apenas a última imagem necessária à recuperação do banco em caso de falha, evitando escritas de dados desnecessários. Para tanto, ETERNAL garante que as transações são primeiramente confirmadas e persistidas em um armazenamento intermediário persistente de NVRAM. ETERNAL usa a baixa latência de escrita aleatória e endereçabilidade por byte da NVRAM para atualizar individualmente a última imagem escrita de um dado em NVRAM. Assim, há uma redução no número de registros de log para várias atualizações sobre um mesmo dado. Adicionalmente, ETERNAL permite o despejo assíncrono do conteúdo da NVRAM para HDD, fazendo com que a etapa de confirmação das transações não fique diretamente dependente da latência de acesso ao disco. Nos experimentos, ETERNAL apresentou um *throughput* significativamente maior que a abordagem baseada em WAL. Os experimentos mostraram ainda que, em ETERNAL, em nenhum momento as transações tiveram que ser suspensas para a realização de despejos para disco.

Em suma, as principais contribuições deste trabalho são:

- Uma arquitetura de persistência híbrida com NVRAM e HDD cujo caminho crítico da etapa de confirmação das transações dependa apenas de escritas em NVRAM;
- Uma estratégia que leva em conta o *skew* de cargas de trabalho OLTP e minimiza o uso de armazenamento NVRAM, diminuindo a necessidade de despejos para disco;
- Um mecanismo de despejo assíncrono do conteúdo da NVRAM para disco que não impacta diretamente o desempenho geral do sistema.

Como resultado das pesquisas realizadas na avaliação da hipótese discutida acima, um trabalho foi submetido à comunidade científica e aceito para publicação. Detalhes do trabalho encontram-

se listados na Tabela 2 abaixo:

Tabela 2 – Publicação.

Título	Autores	Conferência	Situação
ETERNAL: Uma estratégia eficiente de tolerância a falhas utilizando memória não-volátil (GOMES *et al.*, 2019)	**Davi Gomes**, Angelo Brayner, Javam Machado	SBBD 2019	Publicado

Fonte: elaborada pelo autor.

1.2 Organização do Texto

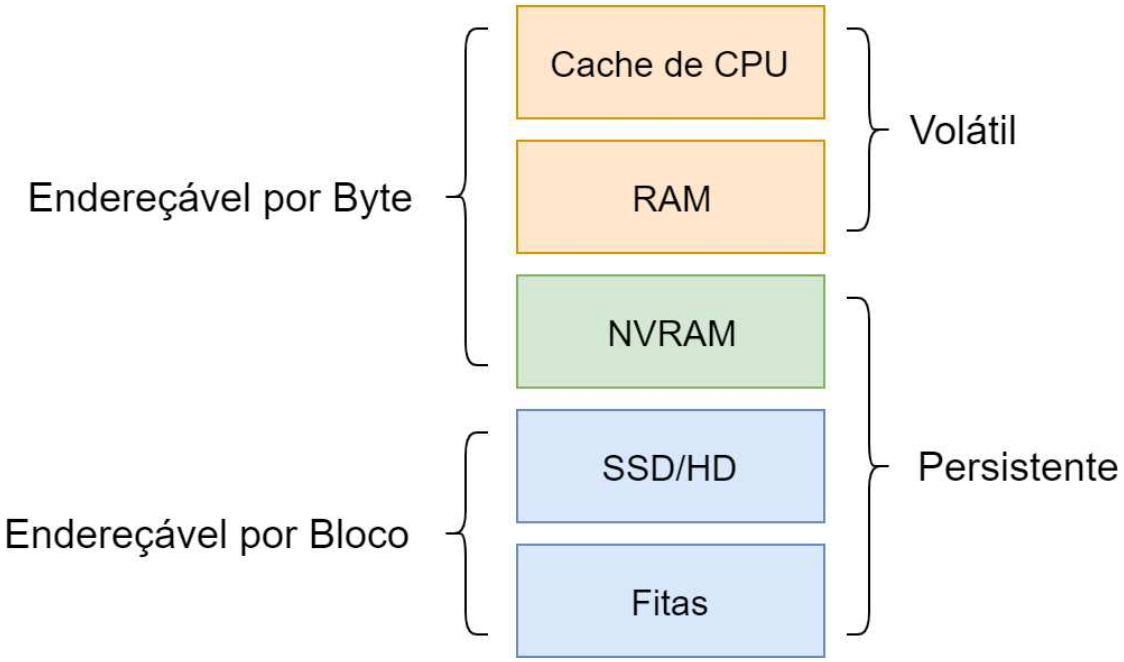
Esta dissertação encontra-se organizada de tal forma que o Capítulo 2 descreve a fundamentação teórica necessária para a compreensão deste trabalho. O Capítulo 3 descreve os trabalhos que de alguma forma se relacionam com ETERNAL. O Capítulo 4 descreve em detalhes o funcionamentos dos protocolos e dos componentes de ETERNAL. No Capítulo 5 são discutidas as avaliações experimentais realizadas e os seus resultados. Finalmente, no Capítulo 6, são sumarizadas o as conclusões deste trabalho e são propostos alguns trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Hierarquias de Memória

Tradicionalmente, os meios de armazenamento nos computadores são diferenciados seguindo uma hierarquia, conforme apresentado na Figura 1.

Figura 1 – Hierarquia de dispositivos de armazenamento.



Os meios de armazenamento possuem algumas características especiais nas quais são agrupados, em relação à volatilidade e acesso a dados.

A volatilidade dos dados é definida como se os dados ainda estivessem presentes quando não há energia para alimentar o hardware, eles são divididos em duas categorias:

- **Volátil.** Armazenamento volátil significa que não mantém dados quando a energia está desligada. Mais especificamente, eles não preservam o estado.
- **Persistente.** Armazenamento persistente significa manter os dados quando a energia está desligada. Mais especificamente, eles preservam o estado e podem armazenar dados.

O acesso a dados é definido como a forma como os dados serão acessados, eles são divididos em duas categorias:

- **Enderecável por Bloco.** Se os dados forem acessados em uma determinada posição, um bloco inteiro contendo essa posição deverá ser lido e carregado.

• **Endereçável por Byte.** Se os dados são acessados em uma determinada posição, eles são lidos a partir dessa posição e os bytes exatos para os dados são recuperados.

Iniciamos a discussão partindo do meio com maior capacidade e menor velocidade, seguindo para armazenamentos menores e mais rápidos.

2.1.1 Fita

O armazenamento em fita não é usado para armazenar dados em sistemas ativos, mas para armazenamento de dados a longo prazo, como backups. As fitas podem armazenar centenas de terabytes em uma unidade, mas o acesso aos dados é sequencial, serial e endereçado a bloco, o que significa que, para uma leitura de ponto, também é necessário recuperar mais dados. Qualquer acesso aleatório exige muito esforço, pois a fita deve ser rebobinada para a posição, um processo que pode levar dezenas de segundos. Portanto, os aplicativos onde essas mídias de armazenamento são usadas devem ser adequados às limitações desse meio. Um exemplo de aplicativo que tira proveito dessas restrições é um aplicativo de log, no qual os dados são adicionados ao longo do tempo e, se for necessário recuperá-los, isso é feito do início ao fim. Financeiramente, as fitas são o meio de armazenamento mais barato, levando em consideração a relação preço por armazenamento.

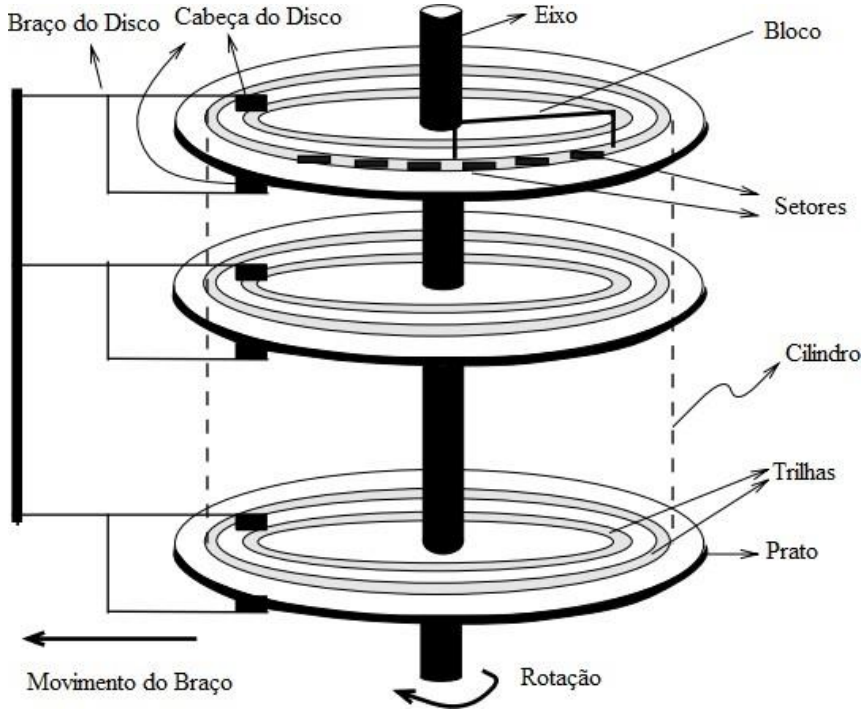
2.1.2 Discos Magnéticos

As unidades de disco rígido são usadas como principal mídia de armazenamento na maioria dos sistemas de computadores até o momento. Eles são compostos de discos magnéticos empilhados e cabeças conectadas que lêem e gravam nesses discos, movidas pelos braços. Os discos rígidos oferecem uma boa relação entre capacidade de armazenamento e velocidade de acesso. Atualmente, os discos rígidos podem armazenar unidades de terabyte em uma unidade e acessá-las em dezenas de milissegundos.

O modo ideal de acesso ainda é sequencial, pois o movimento dos braços é unidirecional, mas, diferentemente das fitas, é possível acessar aleatoriamente em tempos razoáveis, porque, à medida que os discos giram em alta velocidade, a espera máxima por uma seção da cabeça posicionada é uma rotação do disco. O acesso aos dados nas unidades de disco rígido também é feito por bloco, portanto, é importante considerar quais dados serão armazenados no mesmo bloco. É nesses dispositivos que reside a maioria dos dados SGBD tradicionais. As unidades de disco rígido são economicamente viáveis para armazenamento e oferecem a melhor

relação preço por armazenamento e velocidade de acesso. A figura 2 (RAMAKRISHNAN; GEHRKE, 2003) apresenta os componentes internos de um disco rígido.

Figura 2 – Componentes internos de um HD.

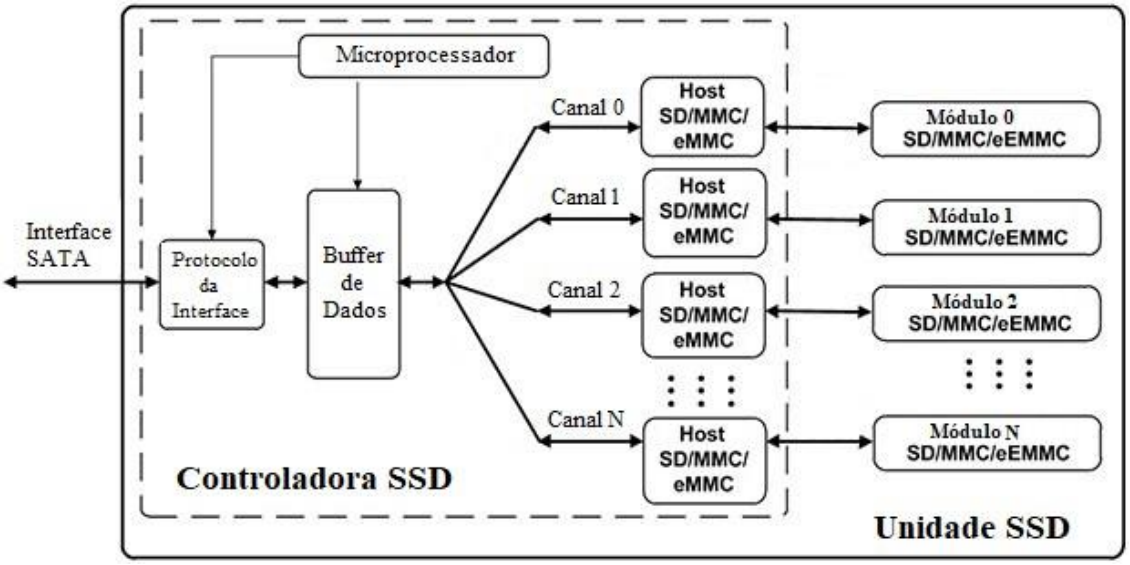


2.1.3 Unidade de Estado Sólido (SSD)

Com a evolução das tecnologias de memória FLASH, os SSD tornaram-se economicamente viáveis. Embora os SSDs ainda tenham armazenamento endereçado em bloco, o acesso aos dados não é feito mecanicamente, mas por impulsos elétricos. Um SSD consiste em várias células de memória FLASH e um controlador, que implementa vários algoritmos, como nivelamento de desgaste, para evitar a rápida degradação das células, uma limitação da tecnologia. Quanto à velocidade de acesso, é possível afirmar que um SSD é pelo menos 10x mais rápido que um disco rígido, dependendo dos dispositivos que estão sendo comparados e como eles estão sendo comparados. Os SSDs não são recomendados para armazenamento de dados a longo prazo, nem aplicativos com uso intenso de gravação devido à sua degradação. Os SSDs disponíveis comercialmente têm capacidade de armazenamento de centenas de gigabytes. Assim, seu alto desempenho faz do SSD a melhor escolha para a última camada persistente de dados. Comparado à RAM, os SSDs têm cerca de 100 vezes a sua latência. Os SSDs são mais caros que os discos rígidos e incorrem em um custo mais alto devido à sua menor durabilidade. A figura 3 apresenta os componentes internos de uma unidade de estado

sólido. Os comandos de I/O vêm da interface SATA e são distribuídos pelo controlador em diferentes canais, facilitando o acesso aleatório e paralelo.

Figura 3 – Componentes internos do SSD



2.1.4 Memória Não Volátil de Acesso Aleatório (NVRAM)

Num futuro próximo, haverá memórias não voláteis. Eles prometem velocidade de acesso próxima aos números da memória principal, sem sua principal desvantagem, a volatilidade dos dados, por serem mídias de armazenamento duráveis. Em contraste com a mídia de armaze- namento apresentada acima, as memórias não voláteis possuem armazenamento endereçável por byte, o que significa que os dados obtidos a partir deste meio de armazenamento são obtidos por si só, utilizando apenas o ponteiro indireto. Existem várias maneiras de implementar esse novo paradigma: a memória de mudança de fase (PCM) (LEE *et al.*, 2010), usa uma estrutura de cris- talização de elementos químicos para caracterizar o estado de um byte; RAM magnetoresistivo (Rachel Courtland, 2014), usa elementos magnéticos em vez de corrente elétrica; memristors memristor, um novo elemento fundamental dos circuitos, capaz de manter o último estado mesmo sem corrente elétrica. A capacidade de armazenamento de destino dessas memórias é equivalente aos SSDs atuais. NVMs, no entanto, têm uma latência de 4x a latência na DRAM, que é bastante próxima, em comparação com os SSDs. Como ainda estão em desenvolvimento por seus fabricantes, só é possível experimentá-lo por meio de simuladores, mas eles já são um objeto de estudo na academia.

2.1.5 Random Access Memory (RAM)

Também chamada de memória principal, a memória de acesso aleatório é onde reside a maioria dos dados que estão em uso no computador. Essas memórias têm velocidade de acesso rápida e capacidade razoável, na ordem de dezenas ou centenas de gigabytes. No entanto, essas memórias são voláteis, dependendo da energia elétrica para manter seu estado. O acesso aos dados é endereçado por byte, permitindo assim a recuperação oportuna dos dados solicitados, quase independentemente da localização desses dados na memória. As aplicações geralmente são projetados para usar a memória principal ao acessar os dados atuais. Nos SGBDs tradicionais, basicamente toda a aplicação é mantida na memória, bem como estruturas auxiliares, por exemplo índices, dados mais recentemente utilizados armazenados em buffers, entre outros. Esses SGBDs foram projetados com pouca RAM disponível; portanto, o gerenciador de buffer precisa de algoritmos específicos para lidar com quais dados devem estar no buffer. A volatilidade dessas memórias gera a necessidade de um controle de operações nos dados residentes, o chamado log. Por esse log, quando há uma falha, o banco de dados pode ser restaurado para um estado consistente, mesmo que os dados mais recentes não tenham sido mantidos. Apesar da recente tendência de queda dos últimos anos, a RAM ainda tem um alto custo em comparação com outras mídias de armazenamento, uma vez que é necessária uma infraestrutura especial para usar grandes quantidades.

2.1.6 Caches de CPU

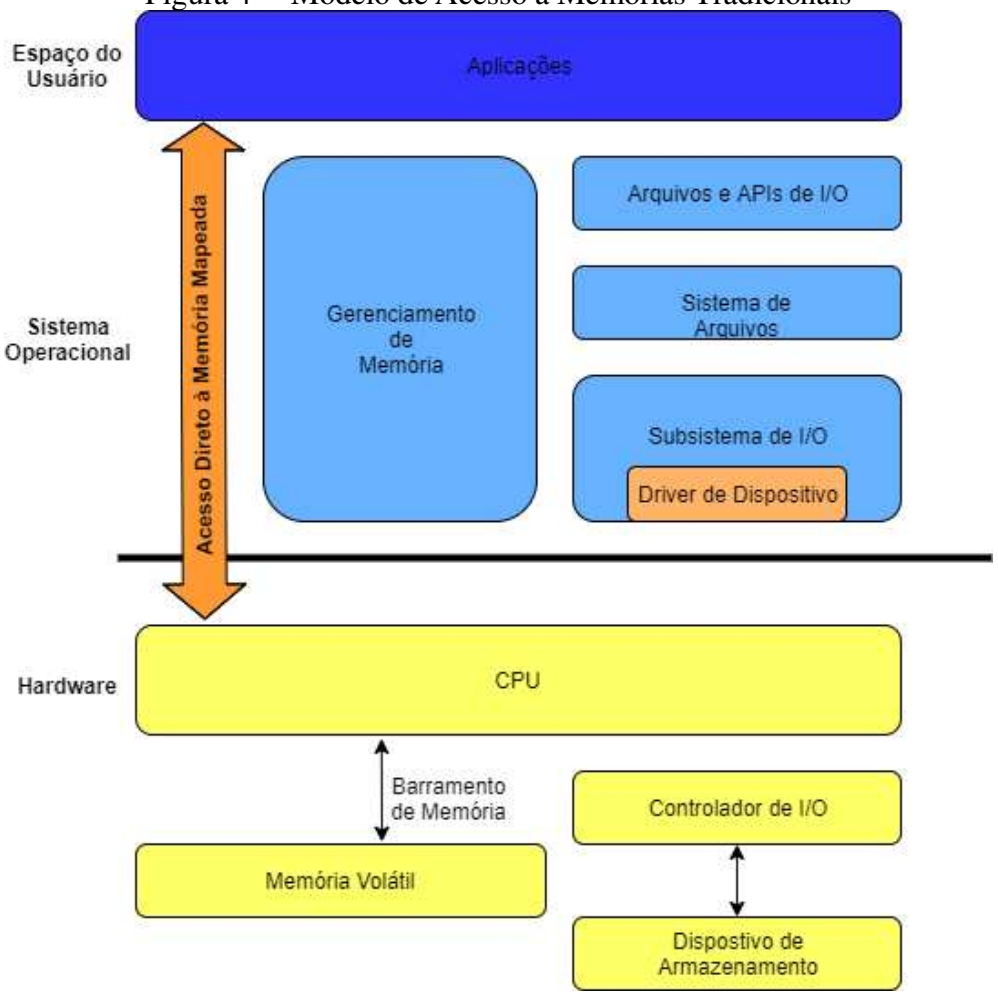
Finalmente, a última camada de memória, a mais próxima do processador, os caches da CPU são memórias especiais de desempenho muito alto e, por isso, de tamanhos muito pequenos. Um cache de CPU típico hoje possui 3 camadas. A primeira camada, na verdade próxima aos registradores, armazena cerca de 256 KB; a segunda, que serve como suporte para a primeira, geralmente 1 ou 2 MBs; e a terceira, que serve como repositório dos dados mais usados da memória, é de aproximadamente 8 MB. A velocidade dessas camadas também é bem diferente. O processador é responsável pela movimentação de dados entre os cache de RAM e CPU, bem como os movimentos entre as camadas.

2.2 Modelo de Programação em NVRAM

Tipicamente, a memória principal volátil é conectada diretamente à CPU por meio

de um barramento de memória. O sistema operacional gerencia o mapeamento de regiões de memória diretamente no espaço de endereço de memória visível das aplicações. O armazenamento persistente, que geralmente opera em velocidades muito mais lentas que a CPU, está conectado por meio de de um controlador de entrada e saída (I/O). O sistema operacional lida com o acesso ao armazenamento persistente tradicional por meio de drivers de dispositivos presentes no sistema operacional. Essa organização pode ser visualizada na Figura 4.

Figura 4 – Modelo de Acesso à Memórias Tradicionais



A combinação do acesso direto das aplicações à memória volátil, juntamente com o acesso de I/O do sistema operacional aos dispositivos de armazenamento, suporta o modelo de programação mais comumente utilizado em aplicações tradicionais, incluindo SGBDs. Nesse modelo, os desenvolvedores alocam estruturas de dados e as operam com granularidade de bytes na memória. Quando o aplicativo deseja salvar dados, ele usa chamadas padrões do sistema de arquivos para gravar os dados em um arquivo previamente aberto. Dentro do sistema operacional, o sistema de arquivos executa essa gravação executando uma ou mais operações

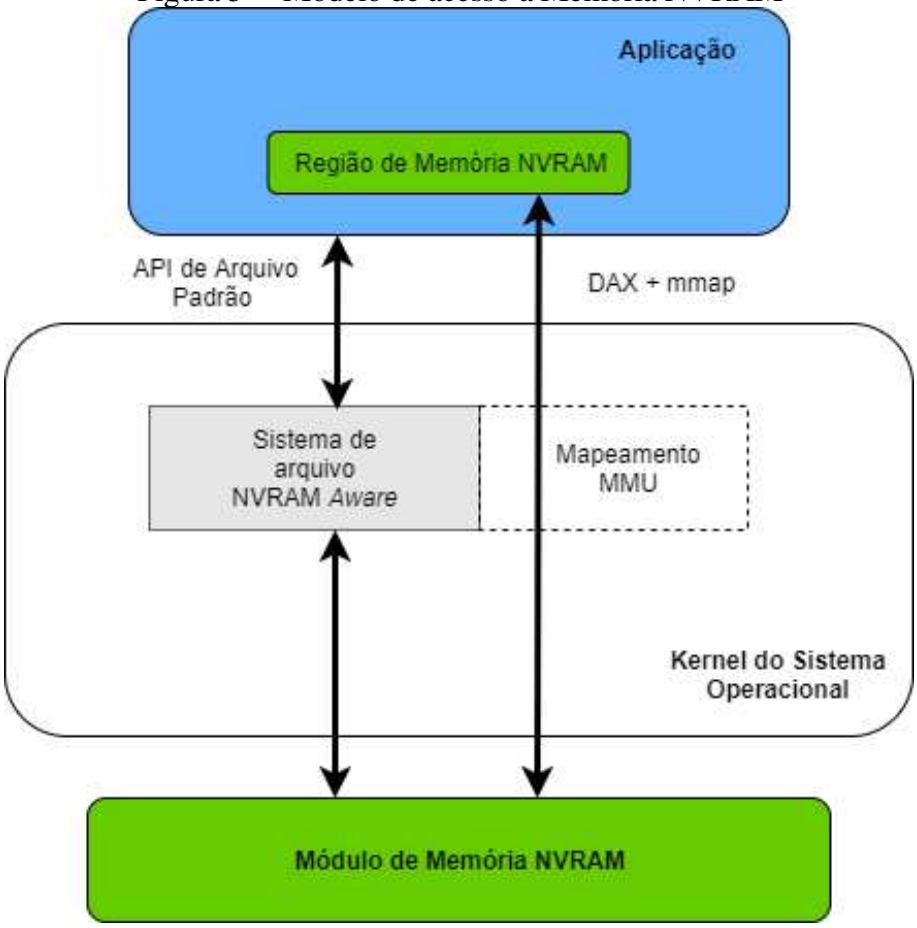
de I/O no dispositivo de armazenamento. Como essas operações de I/O geralmente são muito mais lentas que as velocidades da CPU, é típico para o sistema operacional suspender a aplicação a até que a operação de I/O seja concluída.

Como a NVRAM, assim como memória RAM tradicional, pode ser acessada diretamente pelas aplicações e pode persistir dados diretamente, ela permite que os sistemas operacionais suportem um modelo de programação direta. Os projetistas dos principais sistemas operacionais, Microsoft Windows e Linux, colaboraram na SNIA (Storage and Networking Industry Association) para definir um modelo de programação unificado. Ele é especificado no Modelo de Programação SNIA NVM (SNIA, 2015).

A especificação do modelo de programação descreve várias maneiras pelas quais as aplicações podem usar NVRAM junto com extensões do sistema operacional. Dentre as extensões, destaca-se o gerenciamento da memória não volátil por meio de arquivos. Dentre as vantagens de se utilizar arquivos para gerenciar o armazenamento NVRAM, é possível destacar o uso das funcionalidades providas pelos principais sistemas de arquivos existentes. Por meio deles, por exemplo, é possível organizar, gerenciar, nomear e limitar o acesso de usuários a arquivos e diretórios em NVRAM. É possível também aplicar permissões do sistema de arquivos e gerenciar direitos de acesso para proteger dados armazenados, bem como compartilhá-los com outros usuários. Para tanto, se faz necessário o uso de sistemas de arquivos NVRAM *aware*. Esses sistemas permitem que pequenas gravações de dados sejam executadas mais rapidamente do que em sistemas tradicionais para dispositivos de acesso por bloco, que exijam que o sistema de arquivos leia uma quantidade de memória do tamanho do bloco nativo do dispositivo, modifique o bloco e, em seguida, escreva o bloco completo de volta no dispositivo.

Outra possibilidade é aproveitar as APIs existentes de mapeamento de memória justamente com a funcionalidade Direct Access (DAX) fornecida pelos sistemas operacionais, dessa forma, aplicações que atualmente usam arquivos mapeados na memória podem usar a memória persistente diretamente sem modificações. Depois que um arquivo em NVRAM é criado e aberto, a aplicação pode chamar *mmap* (KERRISK, 2010) nesse arquivo para obter um ponteiro para o início da região de memória persistente. A diferença, conforme mostrado na Figura 5, é que o sistema de arquivo NVRAM *Aware* reconhece que o arquivo está na memória persistente e programa a unidade de gerenciamento de memória (MMU) na CPU para mapear a NVRAM diretamente no espaço de endereço da aplicação.

Figura 5 – Modelo de acesso à Memória NVRAM



A vantagem da abordagem DAX + *mmap* é que nenhuma cópia na memória do kernel é necessária, bem como nenhuma sincronização com o dispositivo através de operações de I/O. A aplicação pode usar o ponteiro retornado pelo *mmap* para operar seus dados diretamente na memória persistente. Como não são necessárias operações de I/O do kernel e como o arquivo é completamente mapeado na memória da aplicação, ela pode manipular grandes coleções de objetos de dados com maior desempenho e de maneira mais consistente se comparado ao gerenciamento de arquivo tradicional. Considerando todas as vantagens acima, esse foi o modelo escolhido para ser utilizado em ETERNAL por meio da biblioteca libpmemobj. A biblioteca fornece acesso à NVRAM por meio do mecanismo descrito acima, além de fornecer abstrações úteis como objetos persistentes e alocadores NVRAM *aware.*

2.3 Transações

Uma transação é a unidade básica de mudança que é executada dentro de um SGBD. Transações consistem em uma série de operações de escritas e leituras. A transação

T1, representada na Figura 6, por exemplo, pode representar uma transferência bancária de 100 unidades monetárias da conta B para a conta A.

Figura 6 – Exemplo de transação

```
1. início T1
2. ler(A)
3. A := A + 100
4. escrever(A)
5. ler(B)
6. B := B - 100
7. escrever(B)
8. fim T1
```

SGBDs são capazes de executar várias transações de maneira concorrente e não raro duas transações precisam escrever um mesmo conjunto de dados no banco. Durante a execução dessas transações, é possível ainda que falhas ocorram e o banco tenha que ser reinicializado. Diante da realidade desses cenários, é papel do SGBD garantir quatro propriedades importantes: Atomicidade, Consistência, Isolamento e Durabilidade. Também conhecidas como *ACID*.

2.3.1 Atomicidade

A propriedade da atomicidade garante que todos os passos executados por uma transação devem se comportar como um passo único e indivisível. Como uma transação é indivisível, ou ela é executada na sua totalidade ou não simplesmente não é. Portanto, se uma transação começar a ser executada, mas falhar por qualquer motivo, quaisquer alterações no banco de dados que a transação possa ter feito devem ser desfeitas. Este requisito deve ser mantido independentemente de a transação ter sido interrompida por qualquer tipo de de falha como um desligamento do sistema, um encerramento inesperado do processo do SGBD, uma falha no sistema operacional etc. Garantir atomicidade pode ser especialmente complicado devido às diferenças entre os diferentes tipos de armazenamentos que tipicamente compõe um SGBD.

Na Figura 6 a importância da Atomicidade pode ser ilustrada no caso de uma interrupção da transação ocorrer logo após a escrita e a subsequente persistência do valor atualizado da conta A na instrução de número 4. Nesse caso o banco ficaria em um estado onde 100 unidades monetárias foram creditadas à conta A, mas nenhuma unidade monetária foi debitada da conta B. Ou seja, dinheiro foi criado indevidamente, levando o banco à um estado

inconsistente.

2.3.2 Consistência

A propriedade da consistência garante que uma transação executada isoladamente, de maneira atômica e partindo de um estado consistente do banco, deve necessariamente deixar o banco em um estado consistente. Essa propriedade não significa apenas garantir restrições tais como as de chave primária ou chave estrangeira, mas significa também garantir regras de negócio relacionadas ao domínio da aplicação. Isso significa que garantir a consistência de uma transação não é apenas responsabilidade do SGBD, pois é responsabilidade dos usuários que escreveram as transações garantir que elas obedecem as regras de negócio da aplicação.

No exemplo da Figura 6, cabe ao desenvolvedor da aplicação garantir, por exemplo, que em uma transferência bancária, o valor debitado em uma conta seja o mesmo creditado em outra, pois essa é uma lógica inerente a esse tipo de aplicação.

2.3.3 Isolamento

A propriedade do isolamento garante que, embora várias transações possam ser executadas simultaneamente, cada transação não tem conhecimento de outras transações sendo executadas, ou seja, suas ações não podem parecer separadas à outras operações do banco de dados que não fazem parte da transação. Mesmo uma única instrução SQL pode envolver muitos acessos separados no banco de dados, e uma transação pode consistir em várias instruções SQL. Portanto, o SGBD deve executar ações especiais para garantir que as transações funcionem corretamente sem interferência da execução simultânea de instruções do banco de dados.

Do exemplo da Figura 6 a importância da propriedade do Isolamento pode ser ilustrada no caso de duas transações realizarem transferências para uma mesma conta. Caso ambas as transações leiam o mesmo valor de A no passo 2, ambas as transações incrementarão A em 100 unidades monetárias e em seguida escreverão o valor atualizado. O problema nesse caso é que apenas o valor da transação que escreveu por último será persistido, perdendo a atualização da primeira transação.

2.3.4 Durabilidade

A propriedade da durabilidade garante que depois que a execução da transação for concluída com êxito e depois que o usuário que iniciou a transação tiver sido notificado de a transação ocorreu, então necessariamente nenhuma falha no sistema possa resultar na perda de dados correspondentes à essa transação. A propriedade da durabilidade garante que, uma vez que uma transação seja concluída com êxito, todas as atualizações realizadas no banco de dados persistam, mesmo se houver uma falha no sistema após a transação concluir a execução.

Do exemplo da Figura 6 todos os passos da transação são executados após a conclusão do passo 8. Porém, como discutido anteriormente, SGBDs são compostos por diferentes meios de armazenamento com diferentes características. Se a transação for confirmada após a execução de todos os seus passos, mas os registros das contas A e B alterados pela transação estiverem presentes apenas em DRAM, então corre-se o risco da violação da propriedade da durabilidade, pois em caso de perda do conteúdo da memória principal, os dados tocados por essa transação também serão perdidos.

2.3.5 Tipos de Falha

A literatura de banco de dados tradicionalmente considera três classes de falhas: falhas de transação, falhas de sistema e falha de meio de armazenamento.

- **Falhas de transação** Ocorre durante o processamento normal, quando uma única transação deve ser revertida devido a conflitos ou bloqueios no protocolo de controle de concorrência, violação de restrições de integridade ou aborto de transação voluntário iniciado pela aplicação ou pelo usuário. O processo de recuperação de falhas de transação é chamado *rollback.*
- **Falhas de sistema** Falhas de sistema geralmente são causadas por uma falha de software ou perda de energia, e o que é perdido - portanto, o que deve ser recuperado - é o estado do processo do servidor na memória principal; isso normalmente implica recuperar imagens de página no *buffer pool*, bem como listas de transações ativas e seus bloqueios adquiridos, para que possam ser abortadas adequadamente. O processo de recuperação de falhas do sistema é também chamado de reinicialização.
- **Falhas de mídia** Em uma falha de mídia, um dispositivo de armazenamento persistente falha, mas o sistema pode continuar em execução, servindo transações que apenas tocam dados no buffer pool ou em outros dispositivos íntegros. O processo de recuperação de

falhas do mídia é chamado de restauração da mídia.

2.3.6 Estados de um Transação

Uma transação que não consegue concluir sua execução com sucesso é denominada abortada. Para garantir a propriedade da atomicidade descrita anteriormente, uma transação abortada não deve manter alterações no estado do banco de dados. Portanto, quaisquer alterações no banco de dados feitas pela transação abortada devem ser desfeitas. Depois que as alterações causadas por uma transação abortada são desfeitas, dizemos que a transação foi revertida.

É parte da responsabilidade do subsistema de recuperação gerenciar o *abort* de transações. Isso é feito normalmente mantendo um log. Cada modificação do banco de dados feita por uma transação é registrada primeiro no log. O identificador da transação que está executando a modificação é registrado juntamente com o identificador do registro de dados que está sendo modificado, com o valor antigo (antes da modificação) e com o novo valor (após a modificação) do registro de dados. Somente então o próprio banco de dados é modificado. Manter um log fornece a possibilidade de refazer uma modificação para garantir atomicidade e durabilidade, bem como a possibilidade de desfazer uma modificação para garantir atomicidade em caso de falha durante a execução da transação.

Uma transação que conclui sua execução com sucesso é considerada confirmada. Uma transação confirmada que executou atualizações leva o banco de dados a um novo estado transacionalmente consistente, que deve necessariamente persistir mesmo se houver uma falha no sistema. Depois que uma transação é confirmada, não é possível desfazer seus efeitos abortando-a.

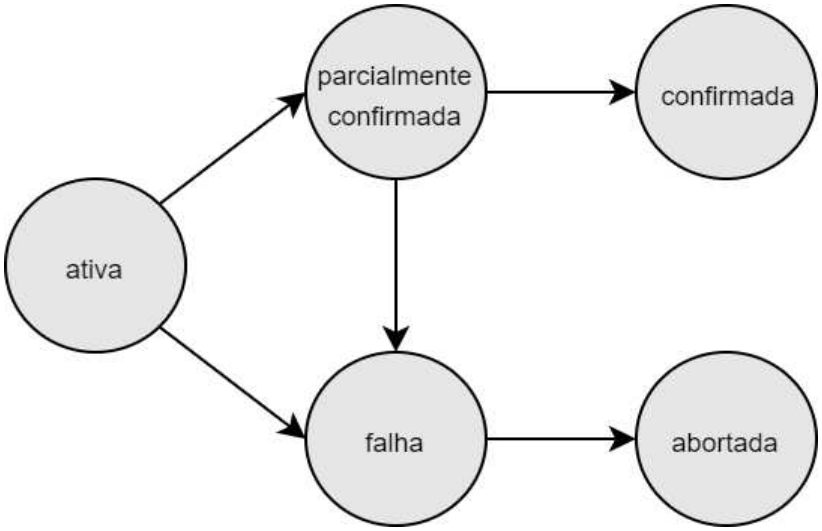
Segundo (SILBERSCHATZ *et al.*, 2020), uma transação deve estar em um dos seguintes estados:

- **Ativa** Estado inicial de uma transação. Ela permanece nesse estado enquanto estiver sendo executada.
- **Parcialmente confirmada** Esse estado é alcançado quando a última operação da transação for executada.
- **Abortada** Estado a partir do qual todas as mudanças ocasionadas pela transação no banco de dados foram revertidas.
- **Confirmada** Estando no qual a transação foi completada com sucesso.

A Figura 7 ilustra os estados de uma transação e as possibilidades de transições entre eles. Toda transação se inicia como ativa e permanece nesse estado até que tenha completado

todas as suas operações ou então até a ocorrência de uma falha. Caso complete todas as suas operações, a transação será considerada parcialmente confirmada, podendo ser confirmada apenas quando for garantindo que o estado transacionalmente consistente do banco após a execução dessa transação possa ser recuperado em caso de falha, o que garante a propriedade da durabilidade. Quando ocorre uma falha, as alterações realizadas por essa transação que tocaram o banco devem necessariamente ser desfeitas, para só então a transação ser considerada abortada, garantindo a durabilidade da transação.

Figura 7 – Estados de uma transação



2.3.7 Write-ahead logging

O conceito de *write-ahead logging* (WAL) foi descrito pela primera vez por (GRAY, 1978) e hoje é utilizado pela grande maioria dos sistemas de banco de dados. Em geral, a ideia por trás do WAL é fornecer atomicidade e durabilidade da transação, mantendo uma estrutura de dados persistente separada, chamada log, além do banco de dados materializado, que é o próprio banco de dados. O termo *write-ahead* refere-se às restrições impostas às gravações realizadas no log e no banco de dados: todos os registros de log pertencentes a um determinado registro de dados devem ser duráveis antes que esse item seja atualizado no banco de dados.

As abordagens baseadas em WAL contrastam diretamente com abordagens que mantenham uma única estrutura de armazenamento persistente. Essas abordagens, de maneira geral, se assemelham à abordagem *shadow paging* (LORIE, 1977). As abordagens *shadow paging* possuem algumas desvantagens. Primeiro, eles são ineficientes com dispositivos de

armazenamento convencionais, como discos magnéticos e unidades de estado sólido, já que a ausência de um log requer o uso de uma estratégia de propagação atômica com uma política de cache *no-steal* e o uso de uma política *force* durante o processamento da etapda de confirmação. Essa configuração pode funcionar razoavelmente bem apenas com grande capacidade de memória principal, pois a política *no-steal* requer que páginas modificadas por transações não confirmadas sejam mantidas em memória volátil, bem como com uma latência de armazenamento muito próxima da memória principal, devido à restrição da política *force*. Segundo, eles limitam severamente a concorrência ao exigir o bloqueio no nível da página. Terceiro, é possível argumentar que elas tendem a ser menos úteis e menos confiáveis por causa da falta de um histórico. O log é uma ferramenta útil para consultas de depuração e auditoria. Além disso, devido ao seu padrão de gravação não destrutivo, o log é mais eficaz no suporte a uma ampla variedade de falhas, incluindo falhas de mídia e de página única. Uma alternativa menos óbvia é eliminar o banco de dados materializado e confiar no log como a única forma de armazenamento persistente.

Atualmente, a grande maioria dos sistemas de banco de dados implementam a abordagem *write-ahead logging* conforme descrita em ARIES (MOHAN *et al.*, 1992a). As duas principais distinções de seus antecessores são o uso de registro fisiológico e o *undo* lógico com compensação. O primeiro permite a independência de recuperação entre objetos, uma característica fundamental. Em essência, cada registro de log descreve alterações em um determinado conjunto de páginas do banco de dados, e é garantido que operações de *redo* tenham seus efeitos restritos apenas a essas páginas. Isso implica que ações de *redo* em páginas diferentes podem ser executadas independentemente.

A outra técnica, *undo* lógico com compensação, determina que desfazer ações deve ser a compensação lógica da ação original. Neste caso, lógico implica granularidade maior que uma página, ou seja, uma tabela, índice ou todo o banco de dados. Por exemplo, uma inserção de um registro em uma tabela é desfeita como a exclusão do mesmo registro dessa tabela, independentemente dos efeitos nas estruturas físicas de dados. Essas ações de *undo* geram registros de log especiais de *redo only* conhecidos como registros de log de compensação, que são processados como qualquer outro registro durante a estapa de *redo*, mas orientam a lógica de desfazer de maneira a garantir a idempotência, ou seja, as ações de recuperação são aplicadas exatamente uma vez, mesmo na presença de falhas no sistema.

2.4 Bancos em Memória Principal

Os dados de um SGBD tradicional geralmente residem em algum meio de armazena- mento durável, também chamado de armazenamento secundário, sejam unidades de disco rígido ou unidades de estado sólido. Conforme descrito na seção anterior, essas mídias de armazena- mento estão em uma magnitude abaixo da memória principal em relação à velocidade de acesso, e os SGBDs tradicionais têm otimizações para não sofrer tanto o impacto da recuperação de dados de um meio de acesso tão lento e com granularidade mais grossa.

Garcia-Molina (GARCIA-MOLINA; SALEM, 1992) descreve em seu trabalho as características de um SGBD otimizado em um cenário em que a RAM, também chamada de memória principal, é usada como o principal meio de armazenamento de dados. São observadas algumas diferenças que esse meio de armazenamento traz para o SGBD:

- **Velocidade de acesso aos dados.** Como a memória principal é mais rápida que o armaze- namento secundário em relação à velocidade de acesso, ela deixa de ser o principal gargalo no processamento de consultas.
- **Volatilidade da memória principal.** Os dados armazenados na memória principal são voláteis, resultando em uma opção de design. Aceite a volatilidade dos dados ou um mecanismo de durabilidade deve ser implementado, como os do DBMS tradicional, no entanto, eles devem restaurar o banco de dados para um estado consistente a partir de uma folha em branco, porque no caso de uma falha, os dados anteriores são perdidos.
- **Fotmato de acesso aos dados.** No armazenamento secundário, recuperar dados espe- cíficos incorre em um alto custo fixo, que é o custo para recuperar um bloco inteiro do armazenamento secundário. Quando os dados residem na memória principal, a CPU pode acessá-los diretamente, devido à memória principal ser endereçável por byte, em vez de endereçável por bloco.
- **Organização dos dados.** A organização dos dados é crucial no armazenamento secundário, porque o acesso é baseado em blocos. Portanto, o acesso a dados em conjunto deve estar em conjunto, porque utiliza o acesso sequencial, que é mais rápido que o acesso aleatório nos discos rígidos e reduz o desperdício de I/O. Na memória principal, essa diferença não é tão grande, porque os dados podem ser acessados através de ponteiros. A organização dos dados ajuda a evitar perseguições desnecessárias de ponteiros.

2.4.1 Armazenamento de Dados e Acesso

Os SGBDs tradicionais em disco geralmente implementam uma série de estratégias dedicadas para melhorar o tempo de resposta quando os dados precisam ser acessados pelo gerenciador de buffer. Como todos os dados em um banco de dados de memória principal já estão no meio de armazenamento mais rápido, o gerenciador de buffer é desnecessário (LEVANDOSKI *et al.*, 2013b).

Embora um banco de dados em um SGBD tradicional possa caber inteiramente no gerenciador de buffer, tornando todas os dados residentes em memória, as estruturas de índice ainda são projetadas para executar o acesso ideal ao disco. A árvore B+, por exemplo, +, não aproveita o acesso endereçado por bytes que o residente da memória principal dados fornece. Os SGBDs em memória principal, por sua vez, implementam estruturas de índice otimizadas para isso, como a Bw-Tree. (WU *et al.*, 2017).

2.4.2 Controle de Concorrência

Como a maior fonte de contenção em SGBDS tradicionais é o acesso ao disco, os protocolos de controle de concorrência pessimistas baseados em bloqueios são aceitáveis, pois não afetam tanto a velocidade de processamento das consultas. Nos SGBDs da memória principal, essa premissa não pode mais ser considerada verdadeira, portanto os protocolos de controles de concorrência que realizam bloqueios são consideravelmente prejudiciais, pois impedem o paralelismo no acesso aos dados. A escolha natural nesse caso é pelos protocolos otimistas, sendo os protocolos de controle de concorrência de múltiplas versões aqueles que se apresentam como um alternativa que evita bloqueios, conforme avaliado por Wu et al. (WU *et al.*, 2017).

2.4.3 Logging e Recuperação

Nos SGBDs tradicionais, enquanto uma transação é executada, ela cria vários registros de log, descrevendo as operações executadas para que, quando ocorrer uma falha, o banco de dados possa ser restaurado para um estado consistente. ARIES (MOHAN *et al.*, 1992b), o protocolo de recuperação mais famoso, usa duas etapas, uma etapa *undo*, que analisa as alterações na memória que não foram confirmadas e uma etapa de *redo*, que repete as alterações confirmadas.

Uma otimização que vale a pena mencionar é chamada de confirmação de grupo

(*group commit*), em que várias transações escrevem suas entradas de log um *buffer* e, se ele for preenchido ou se um certo *timeout* for atingido, é gravado em disco, confirmando várias transações com apenas uma operação de gravação no dispositivo de armazenamento.

Em bancos de dados em memória principal, no entanto, todo o banco em DRAM é perdido após uma falha no banco de dados. Esse fato associado ao esforço necessário para ter o protocolo de log e recuperação motiva alternativas, como o *Command Logging* (MALVIYA *et al.*, 2014) que, em vez de registrar o estado dos dados, registra os comandos executados pelo DBMS e os executa novamente para recuperação. *Write-behind logging*, a estratégia proposta por Arulraj et al. (ARULRAJ *et al.*, 2016b), aproveita as múltiplas versões de registros e as características das memórias não voláteis para permitir que as transações realizem operações *in place* e que loguem apenas um conjunto leve de metadados que permitem descartar versões inconsistentes após uma falha.

Muito embora os dados residam em DRAM nos bancos de dados em memória principal, os registros de log, por definição, não podem estar na memória principal, pois devem ser persistentes, e a operação de confirmação para transações com operações de gravação também depende de este registro sendo gravado no disco. No entanto, gravar registros em disco é muito menos dispendioso do que ler e gravar dados em disco, pois é uma operação exclusivamente sequencial. O tema da recuperação será retomado nos trabalhos relacionados.

2.4.4 Processamento de Consultas

O processamento de consultas por si só não possui muitas alterações ao se comparar SGBDs de memória principal e com aqueles baseados em disco. Já os componentes do processamento da consulta possuem algumas diferenças significativas. Os custos de acesso agora pesam menos na decisão do otimizador. Os otimizadores agora consideram menos custos de acesso e mais custos de processamento. Portanto, algoritmos otimizados para acesso ao disco, como o *sort-merge join*, não são tão eficazes. A compilação de planos de consulta para código de máquina também é uma otimização para SGBDs em memória principal, pois o modelo tradicional de interpretação de consultas acarreta um custo inerente, já a compilação ajuda a evitar indireções desnecessárias.

2.5 Cargas de Trabalho

Aplicações que utilizam bancos de dados relacionais geralmente se enquadram em

uma de duas grandes categorias: Processamento de Transações em Tempo Real (OLTP) , que concentra aplicações com perfil de muitas escritas e consultas pontuais a dados e Processamento Analítico em Tempo Real (OLAP), que concentra aplicações que efetuam leituras e agregações, sobre grandes volumes de dados. Como são perfis de acesso a dados distintos, existem SGBDs que atuam de forma ótima em cada uma das categorias.

2.5.1 OLTP

Cargas de trabalho OLTP são caracterizadas pelo seu grande volume de operações de escrita, assim como operações pontuais, e habitualmente recuperam todos os atributos de um registro. A maioria dos bancos de dados relacionais e aplicações comerciais possuem este perfil. Cargas OLTP são compostas de inserções e atualizações sobre tuplas, além de consultas pontuais, que recuperam tuplas inteiras, com seus resultados baseado na igualdade de chaves.

2.5.2 OLAP

As cargas de trabalho OLAP são caracterizadas por suas operações somente leitura que varrem todas as tuplas para recuperar valores de um atributo. Essas consultas geralmente são executadas sobre toda a tabela. Alguns SGBDs foram projetados especificamente para atender esse perfil de consulta. As cargas de trabalho OLAP são compostas de varreduras e agregações de tabela que recuperam um pequeno subconjunto de atributos de todos os registros, com seus resultados baseados principalmente em grandes intervalos.

2.5.3 Organização dos Dados

Cada carga de trabalho mencionada acima acessa os dados de uma maneira diferente e as organizações de dados podem ser otimizadas para responder melhor a um tipo específico de carga de trabalho. Devido às suas características, as consultas OLTP e OLAP podem ser melhor respondidas por diferentes layouts de dados e pelas possíveis otimizações permitidas em cada layout. O armazenamento em linhas é melhor para consultas OLTP, pois todos os dados pertencentes a uma tupla são armazenados juntos, enquanto o armazenamento de colunas é excelente em cargas de trabalho OLAP, porque suas consultas se preocupam mais em recuperar o mesmo atributo de vários registros.

Abadi et al. (ABADI *et al.*, 2008) conduziram um estudo em que um SGBD de

armazenamento de colunas nativo é comparado com um SGBD de armazenamento baseado em linhas altamente otimizado para consultas OLAP. Eles descobriram que, mesmo com toda otimização em vigor, como particionamento vertical ou índices em uma coluna completa, o armazenamento de colunas nativo ainda teria um desempenho superior a um SGBD baseado em linhas, justificando a separação entre layouts e tipos.

2.6 Temperatura de Dados

Muitos trabalhos (FUNKE *et al.*, 2012) (ARULRAJ *et al.*, 2016a) (LANG *et al.*, 2016) dão uma definição clara da temperatura dos dados, pelo menos a definição de dados quentes e frios. Dados quentes são os registros que estão sendo acessados atualmente pela carga de trabalho, geralmente para consultas de atualização e são principalmente registros recentes. À medida que esses registros envelhecem, sua taxa de acesso diminui e eles ficam frios. Dados frios são os registros que não estão sendo acessados no momento e, eventualmente, são acessados para consultas de varredura e agregação, que podem ser otimizadas para evitar a recuperação de registro por registro, mantendo alguns valores pré-calculados. A temperatura dos registros em uma tabela é inerentemente dependente da carga de trabalho, dependendo de quais registros são mais lidos ou atualizados.

3 TRABALHOS RELACIONADOS

Nesta seção serão apresentados os trabalhos relacionados ao tema de persistência e recuperação de falhas em sistemas de bancos de dados em memória. Mais especificamente serão descritos e analisados os trabalhos recentes sobre recuperação em sistemas de memória principal e técnicas de recuperação utilizando memória não volátil.

3.1 Recuperação em Bancos de Dados em Memória Principal

O log em sistemas de memória principal é otimizado para alto rendimento e baixa latência. Como a I/O do log é o principal gargalo, esses sistemas tentam reduzir o volume de log o máximo possível, mais do que sistemas baseados em disco (FAERBER *et al.*, 2017). Nesta seção serão discutidas as abordagens Command Logging e Silo.

3.1.1 Command Logging

Em seções anteriores foram discutidas técnicas de recuperação baseadas em log fisiológico, uma vez que é a estratégia adotada pelo ARIES. No entanto, alguns projetos recentes de sistemas de banco de dados em memória propõem uma forma extrema de log lógico chamado *command logging* (MALVIYA *et al.*, 2014), cujo objetivo principal é reduzir a sobrecarga do processo de log para um mínimo. Há um *trade-off* em *logging* e recuperação em que as operações podem ser registradas em níveis mais altos de abstração, como, por exemplo, uma instrução SQL, que produz registros mais compactos e de maior granularidade (HAERDER; REUTER, 1983). No entanto, isso implica custos de *checkpoints* e esquemas de propagação mais caros ou complexos, porque o banco de dados materializado deve ser mantido em um estado de consistência compatível com a granularidade das operações registradas. Por exemplo, se as operações registradas forem instruções SQL, o banco de dados materializado deverá ser mantido atômico em relação à execução dessas instruções SQL. Isso significa que uma tupla, por exemplo, não poderá ser inserida em uma tabela se um registro correspondente não estiver inserido em um índice secundário ou se um determinado gatilho pós-inserção não é executado completamente. A introdução do *command logging* no sistema de banco de dados H-Store (MALVIYA *et al.*, 2014) foi motivada pela observação de que a inserção de registros no log representava aproximadamente 30% da sobrecarga da CPU em uma carga de trabalho OLTP típica (HARIZOPOULOS *et al.*, 2008). No *command logging*,

cada registro de log descreve uma única transação, codificada como um identificador de procedimento armazenado juntamente com os argumentos usados para invocá-lo. Como isso é muito mais compacto do que produzir um registro fisiológico para cada operação de uma transação, os autores esperam que a maioria dos 30% dos ciclos de CPU mencionados acima possam ser salvos e gastos na execução de transações.

Um aspecto geralmente negligenciado do *command logging* é que ele não elimina a sobrecarga de gerar registros fisiológicos, pois esses ainda são necessários para o *roll back* de transações. Se uma transação precisar ser revertida no meio de sua execução, o sistema de banco de dados deverá utilizar logs físicos ou fisiológicos em memória principal para desfazer suas operações. Portanto, a sobrecarga do log eliminada com o *command logging* é a inserção desses registros em um *buffer*de log centralizado e sua persistência no momento da confirmação. Finalmente, o *command logging* é uma abordagem menos flexível para aplicações, pois, além do requisito de que toda transação deve ser um procedimento armazenado, requer execução determinística da transação (THOMSON; ABADI, 2010). Como as transações confirmadas são literalmente reexecutadas durante a recuperação, a manutenção de um estado consistente do banco exige que elas produzam exatamente os mesmos efeitos que na execução original. Isso impede o uso de entradas externas ou aleatórias como argumentos de uma transação, pois o banco precisa ser recuperado para o mesmo estado consistente que se encontrava antes da falha. Entradas desse tipo podem causar a execução não determinística da transação. Essa restrição é difícil de verificar automaticamente na prática.

3.1.2 Silo

Silo (TU *et al.*, 2013) é um banco de dados em memória principal de alto desempenho e construído sobre uma Masstree (MAO *et al.*, 2012). Silo suporta transações na forma de *stored procedures*. Arquiteturalmente, uma tabela Silo consiste em um conjunto de registros na memória indexados em uma chave primária, juntamente com qualquer número de índices secundários. Silo foi projetado para ser escalável em grandes máquinas com vários núcleos. Silo garante a atomicidade e a durabilidade de suas transações por meio do uso de logs distribuídos em diversos discos. Assim como ocorre na maioria dos bancos de dados em memória, Silo não realiza operações de *undo*. O seu log, portanto, é do tipo *redo only*.

A chave para o desempenho e a escalabilidade do Silo é que ele reduz as gravações em *hotspots* de memória compartilhada, evitando pontos de contenção entre os múltiplos cores.

Em vez de atribuir a uma transação um *timestamp* exclusivo, o que demandaria a sincronização da atribuição do *timestamp*, Silo usa uma abordagem baseada em épocas em que todas as transações leem um número de época global que é incrementado periodicamente. Silo fornece transações serializáveis por meio do rastreamento dos conjuntos de leitura e do armazenamento dos conjuntos de escrita das transações em um buffer. Ao confirmar, uma transação adquire todos os bloqueios de registros em seu conjunto de gravação, gera seu ID lendo a época global atual, valida seu conjunto de leitura ou aborta caso a validação falhe. No final, ela escreve no banco o seu conjunto de gravação junto com seu identificador de transação, liberando os bloqueios de gravação.

O silo explora o paralelismo multicore ao longo de seu projeto de durabilidade e recuperação (ZHENG *et al.*, 2014). Silo executa o seu log *redo only* em paralelo em vários discos, garantindo que as atualizações de uma época sejam duráveis antes das atualizações de épocas com valores maiores. A recuperação é paralelizada, iniciando a partir do carregamento dos arquivos de *checkpoint* em paralelo para reconstruir o estado na memória, seguido pela repetição paralela do log para trazer o banco de dados para um estado consistente.

3.2 Recuperação Utilizando Memória Não-Volátil

A diferença nas características de desempenho entre armazenamentos voláteis (DRAM) e não voláteis (HDs, SSDs) influencia em grande parte o projeto do SGBD. Todavia a chegada do armazenamento NVRAM, quase tão rápido como DRAM e com mesma granularidade de acesso, requer a revisão de muitas dessas escolhas de projeto. Serão apresentados a seguir alguns trabalhos que propuseram abordagens para o subsistema de *logging* e recuperação utilizando NVRAM.

3.2.1 NV-Logging

O NV-Logging (HUANG *et al.*, 2014) é uma abordagem focada na melhoria do subsistema de log explorando NVRAM. É proposta para bancos de dados tradicionais em disco. Os autores afirmam que usar a NVRAM para log é uma solução mais econômica do que usar a NVRAM no lugar do cache DRAM ou simplesmente usá-la para manter o banco de dados e o log ao mesmo tempo. Isso ocorre porque, em sistemas com memória principal suficiente para conter todo o conjunto de dados em uso, as gravações de log se tornam o principal gargalo da escalabilidade da taxa de transferência de transações. Se esse gargalo for

tratado adequadamente, uma pequena quantidade de NVRAM poderá trazer melhorias substanciais no desempenho.

A abordagem NV-Logging reduz o gargalo relacionado ao log. Em sistemas tradicionais, a calda do log é armazenada em um buffer centralizado em memória principal no qual as diferentes transações concorrentes devem ser cuidadosamente sincronizadas para que possam inserir suas entradas nesse buffer. Essa sincronização se faz necessária porque as entradas não possuem tamanho fixo, fazendo com que transações posteriores dependam do *offset* de transações anteriores para terem suas entradas escritas. Enquanto um buffer de log centralizado ainda é necessário em NV-Logging, o buffer consiste apenas de entradas de tamanho fixo que apontam para o conteúdo real do registro de log na NVRAM. Embora isso não elimine um ponto de contenção global para operações de log, reduz significativamente a sobrecarga da seção crítica em comparação com os buffers de log centralizados. Para oferecer suporte ao *rollback* da transação e liberar os registros de log relevantes no momento da confirmação, cada transação mantém uma lista encadeada de seus registros de log produzidos com ponteiros para a memória alocada no NVRAM. Um possível problema com essa abordagem é que ela transfere parte do ônus do *buffer* centralizado de log para o gerenciador de memória persistente, que precisa fornecer ponteiros persistentes para registros de log de tamanho variável de maneira *thread safe*, de tal maneira que várias *threads* possam alocar regiões de memória persistente de maneira concorrente e correta. Segundo os autores, isso é mitigado pré-alocando todos os objetos de log na NVRAM quando o buffer de entradas de log é alocado. O trabalho conclui que o uso de NVRAM para logging é uma solução eficaz para melhorar o rendimento da transação para cargas de trabalho residentes na memória. Além disso, requer pouca alteração nas bases de código existentes e não implica perda de recursos, desempenho ou funcionalidade de confiabilidade.

3.2.2 FOEDUS

FOEDUS (KIMURA, 2015) é uma *engine* transacional em NVRAM que se baseia em um mecanismo de “dualidade de página”, no qual todas as páginas do banco de dados existem em um *snapshot* de somente leitura ou em um estado modificável de volatilidade. Em vez de acompanhar esses estados com um mapeamento separado, como é feito nas abordagens *shadow paging* por exemplo, FOEDUS usa a Foster B-tree (GRAEFE *et al.*, 2012), onde cada ponteiro da estrutura de dados contém os endereços das versões *snapshot* e volátil. As páginas de *snapshots* residem principalmente em NVRAM, mas podem ser armazenadas em cache na

DRAM, enquanto que as páginas voláteis residem exclusivamente na DRAM. O estratégia de logging em FOEDUS é baseado no Silo, que usa o log de *redo only* distribuído e um mecanismo de épocas para estabelecer a durabilidade das transações. A principal vantagem desse esquema, além da melhor escalabilidade do log distribuído, é que o log contém apenas modificações de transações confirmadas.

FOEDUS explora o log de *redo only* para executar a propagação assíncrona de atualizações nas versões de *snapshot* de cada página. Para isso, ele periodicamente varre esse log, aplicando atualizações à última imagem de *snapshot* de cada página. Para fazer isso de maneira atômica, as atualizações são repetidas em cópias separadas, trocando-se apenas os ponteiros da página no final. O autor chama esse processo de instalação de ponteiros. A desvantagem é que esse processo requer que as transações sejam suspensas por um curto período de tempo, o que inevitavelmente causa picos de latência periódicos. Esse mecanismo de propagação garante que as páginas de *snapshot* estejam sempre em um estado confirmado, mas não no estado confirmado mais recente. Portanto, a recuperação envolve a reprodução do log e a instalação de novos ponteiros, conforme descrito acima, enquanto o sistema não está disponível para novas transações. Para reduzir a latência de *hot spots*, como as páginas mais lidas e os nós raizes dos índices, as páginas de *snapshot* são adicionalmente armazenadas em cache na DRAM.

3.2.3 Write-behind Logging

O *write-behind logging* (WBL) (ARULRAJ *et al.*, 2016c) é uma abordagem baseada na política *force* para o processamento da etapa de confirmação. A bordagem utiliza também armazenamento de múltiplas versões, ou seja, atualizações ocorrem somente com a criação de novas versões, contrastando com atualizações *in place*. Ele aproveita a capacidade de endereçabilidade por byte da NVRAM para usar registros alinhados às linhas de cache da CPU como a unidade de persistência durante a confirmação. Como a persistência depende da política de *flush* de cache da CPU, as atualizações não confirmadas podem atingir o armazenamento persistente e um esquema de log baseado em épocas é usado para determinar as transações perdedoras durante a reinicialização. Cada chamada de confirmação de grupo determina um intervalo de épocas (*cp, cd*), de modo que todas as transações com época menor que cp se tornem duráveis e nenhuma transação com época acima de cd tenha feito alterações no banco de dados. Esse par de valores é gravado no log depois que todas as alterações anteriores ao cp foram persistidas, o que explica o termo *write-behind*. As transações de longa execução que abrangem vários intervalos desse tipo devem ter suas épocas registradas individualmente. A reinicialização na abordagem WBL é muito curta, porque

envolve apenas a determinação dos intervalos (*cp, cd*), que correspondem à determinação do conjunto de transações perdedoras, todas as transações com época neste intervalo são transações perdedoras. Cada falha do sistema adiciona um novo intervalo de transações perdedoras, e a lista atual de intervalos é periodicamente verificada e reconstruída na fase de análise de log. É também necessário um coletor de lixo assíncrono para apagar as modificações das transações perdedoras e excluir as entradas dessa lista, o que também permite o truncamento de log. Para ocultar suas alterações das novas transações pós-falha, o registro de data e hora de cada registro acessado é verificado; se cair em um dos intervalos de transação perdedoras, a versão mais antiga será acessada. Portanto, o mecanismo funciona apenas com armazenamento com múltiplas versões, em que cada registro aponta para sua versão anterior.

Enquanto o esquema WBL diminui significativamente o tempo de recuperação com sua política *force*, a sobrecarga no processamento normal aumenta por vários motivos. Primeiro, enquanto o log atual é realmente muito compacto, o armazenamento de múltiplas versões é essencialmente uma forma incorporada de log, pois as imagens de antes e de depois de cada registro estão sempre disponíveis no banco de dados materializado. Segundo, ao considerar o tráfego de armazenamento volátil para armazenamento persistente, surge o mesmo problema: forçar novas versões de registros requer essencialmente a mesma largura de banda de gravação que forçar refazer registros de log com imagens posteriores. De fato, o último é mais rápido porque é feito sequencialmente, e isso é verdade mesmo quando são consideradas gravações de memória volátil a persistente, graças à localidade do cache da CPU. Os resultados experimentais obtidos pelos autores da abordagem WBL mostram que ela tem um desempenho melhor que a abordagem WAL, ou seja, possui maior *throughput* médio de transações. No entanto, a implementação do WAL também usa armazenamento em várias versões, quando, de fato, o WAL é projetado para estratégias *in place*.

3.3 Comparação com ETERNAL

Abaixo na Tabela 1 é feita uma comparação das principais características de alguns dos trabalhos discutidos com relação à abordagem ETERNAL. O primeiro critério considerado é se o trabalho é pensado para sistemas de banco de dados em memória principal. O segundo critério é se o trabalho tira proveito das características das memória do tipo NVRAM ou se simplesmente as utilizada como um disco com acesso mais rápido. O último critério é se o trabalho utiliza uma arquitetura híbrida de Disco + NVRAM.

Tabela 3 – Comparação de trabalhos relacionados

Trabalhos	*Command Logging*	*Silo*	*NV-Logging*	*FOEDUS*	*Write-behind Logging*	ETERNAL
Projetado para SGBDs em memória	Sim	Sim	Não, SGBD tradicional em disco	Sim	Sim	Sim
Faz uso das características únicas de NVRAM	Não	Não	Sim	Sim	Sim	Sim
Usa armazenamento persistente híbrido com NVRAM e disco	Não, apenas disco	Não, apenas disco	Sim	Não, apenas NVRAM	Não, apenas NVRAM	Sim

Fonte: elaborada pelo autor.

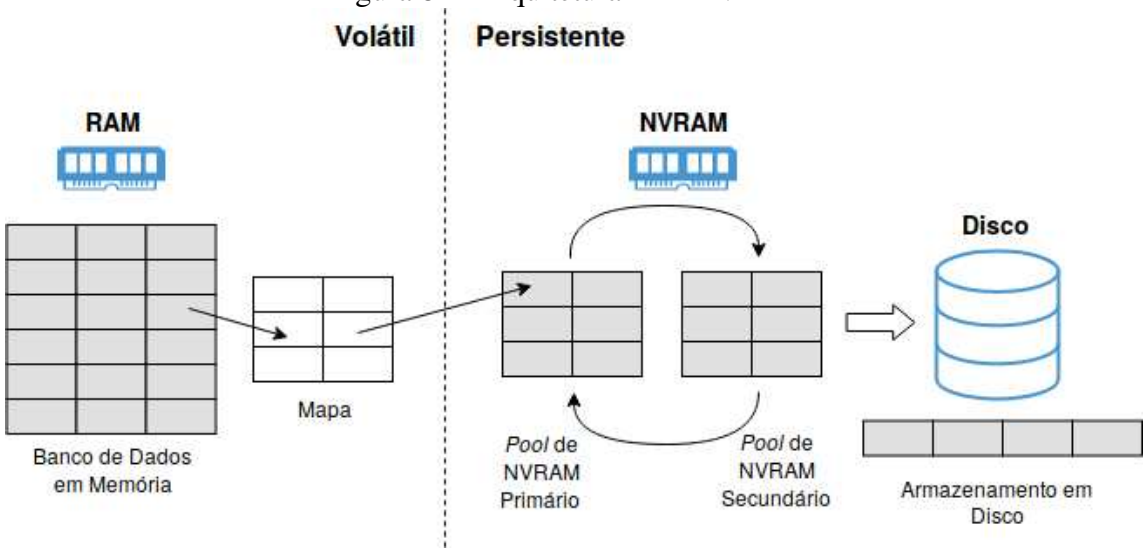
4 ETERNAL

Arquiteturas de SGBDs que visam utilizar NVRAM da melhor maneira possível devem necessariamente fazer uso das suas principais características. Como foi visto, além de possuir uma latência de acesso ordens de magnitude inferior quando comparada a HDDs e SSDs, memórias NVRAM possuem um *gap* entre taxas de transferência de acesso sequencial e taxas de transferência de acesso aleatório muito menor quando comparadas a dispositivos de acesso por bloco.

Arquitetos devem considerar a capacidade de endereçamento por byte das memórias não voláteis, pois esta característica permite o uso de estruturas de dados semelhantes àquelas utilizadas em memória RAM tradicional, evitando a sobrecarga do gerenciamento de páginas. O projeto de ETERNAL, a arquitetura proposta neste trabalho, leva em consideração todas essas características visando fazer o uso ótimo da tecnologia NVRAM.

Como será visto, o uso de NVRAM é imprescindível em ETERNAL, pois, diferente da NVRAM, HDDs e SSDs são endereçáveis por bloco, portanto, os dados devem ser agrupados em páginas, ocasionando *overhead* adicional em memória principal. Atualizações *in place* em dispositivos de bloco podem facilmente alcançar diversas páginas, incorrendo no custo do *flush* de várias páginas em um dispositivo que já possui alta latência. Pior ainda, o padrão de acesso a essas páginas não é necessariamente sequencial, penalizando ainda mais o tempo de acesso em tais dispositivos.

Figura 8 – Arquitetura ETERNAL



A arquitetura do ETERNAL pode ser vista na Figura 8. O banco de dados se

localiza em espaço de armazenamento volátil, materializado pela memória principal, enquanto que o espaço de persistência é composto por NVRAM e HDD. Do ponto de vista do banco, é na memória principal onde são armazenadas as tuplas, os índices, os metadados etc. Já na NVRAM, são armazenados os dados necessários à confirmação das transações. ETERNAL divide a NVRAM em *pool* primário e secundário. O *pool* primário é utilizado diretamente para o armazenamento de dados relativos à etapa de confirmação enquanto o *pool* secundário é utilizado na realização do despejo desses dados para disco. Há também um mapa ligando registros em memória volátil a endereços de memória não volátil. O mapa indexa as imagens presentes no pool, evitando a necessidade de percorrer a lista encadeada presente em NVRAM no momento da confirmação. Além disso, a latência de acesso da RAM é menor que a da NVRAM, logo, indexar as imagens utilizando um mapa em RAM é mais eficiente. O mapa é implementado utilizando uma tabela hash dinâmica, com complexidade média de inserção O(1).

Transações podem realizar dois tipos de operações: leituras e escritas. Inserções, atualizações e exclusões são operações de escrita. Como já foi discutido nas sessões 2.3.4, para garantir a durabilidade de uma transação, é necessário que as escritas realizadas por ela estejam persistidas em armazenamento durável antes da sinalização de sua confirmação. Em ETERNAL, a durabilidade é garantida por meio de seu protocolo de confirmação que utiliza o mecanismo de lista de imagens da ser descrito na próxima sessão. A garantia da durabilidade é sinalizada pela etapa de confirmação das transações bem sucedidas, já a atomicidade consiste na capacidade de recuperar o banco de dados para um estado no qual os efeitos de transações não confirmadas não sejam visíveis.

4.1 Protocolo de confirmação

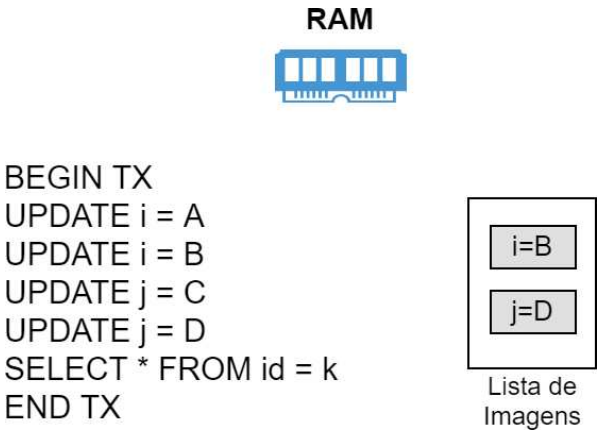
Um importante aspecto de ETERNAL é o seu protocolo de confirmação, ou seja, quais etapas devem necessariamente ser realizadas para que o SGBD possa notificar o usuário que uma transação foi executada com sucesso. Cada etapa deste processo visa garantir a durabilidade das transações.

Para cada nova transação submetida ao SGBD, é atribuída uma estrutura de dados chamada de lista de imagens. Essa estrutura contém a última imagem de cada registro escrito pela transação. Se um registro é escrito mais de uma vez por uma mesma transação, então apenas a sua última imagem desse registro será mantida na lista de imagens. Se uma transação atualiza um mesmo registro três vezes, por exemplo, apenas a imagem da terceira atualização será mantida na lista. Caso uma transação insira um novo registro, então uma nova imagem desse registro será necessariamente inserida na lista de imagens, pois trata-se de um registro

novo, não possuindo uma imagem anterior na lista para ser atualizada.

No exemplo da Figura 9, é possível ver uma transação que atualiza o valor de i e de j duas vezes cada. Nota-se que, para cada registro escrito, apenas o valor da última operação de atualização foi adicionado à lista de imagens. Percebe-se também que o valor de k foi lido, mas não foi adicionado à lista de imagens, pois não foi realizado nenhuma escrita nesse registro. A lista de imagens pode ser implementada utilizando qualquer estrutura de dados que garanta a unicidade de um registro com uma mesma chave, de tal maneira que quando um registro que já exista na lista sofra uma atualização, então ele tenha apenas a sua imagem atualizada para o último valor.

Figura 9 – Funcionamento da lista de imagens

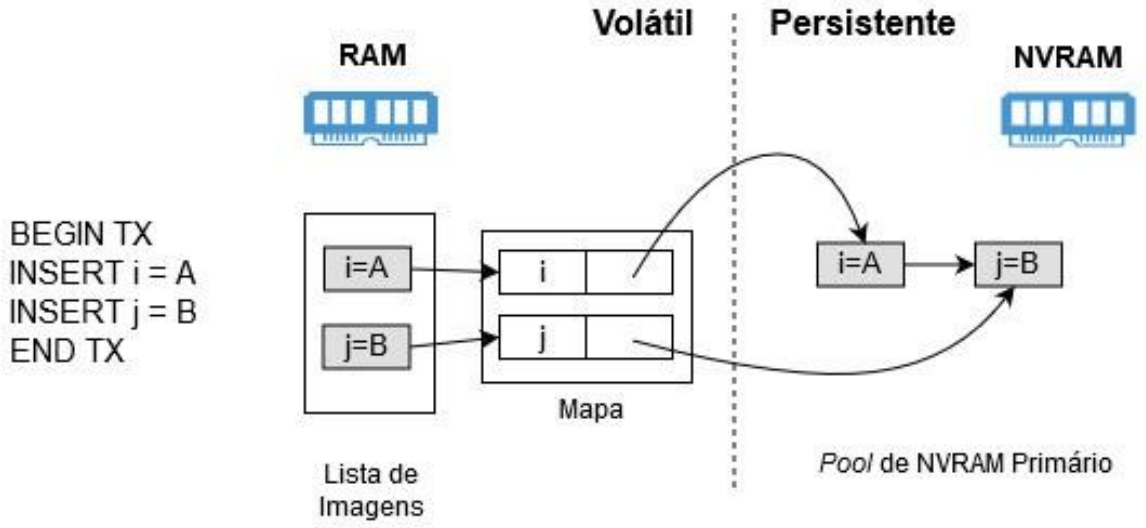


Todo o mecanismo de execução de transações e de preenchimento da lista de imagens é realizado apenas em memória principal. No momento da etapa de confirmação, todos os registros já estarão atualizados no banco materializado em memória principal e todas as últimas imagens escritas pela transação a ser confirmada já estarão presentes na sua respectiva lista de imagens. Somente quando uma transação atingir a sua etapa de confirmação é que ETERNAL irá garantir a persistência do conteúdo de sua lista de imagens de maneira atômica.

A lista será inicialmente persistida no *pool* de NVRAM primário, que é organizado no formato de uma lista encadeada. A maneira pela qual ETERNAL garantirá a durabilidade das transações é por meio da persistência de suas listas de imagens, o que significa que cada transação só poderá ser confirmada após a persistência de sua respectiva lista de imagens. Na Figura 10 é ilustrado o exemplo de uma transação simples que popula o SGBD com seus primeiros dois registros. É possível perceber que sua lista de de imagens possui as duas imagens relativas aos registros inseridos pela transação. Cada imagem presente na lista de imagens gerará uma nova entrada tanto no mapa quanto no *pool* de NVRAM primário. Cada entrada do mapa representa

um par que mapeia o identificador único do registro à um ponteiro persistente, que nada mais é que um endereço de um conteúdo de NVRAM.

Figura 10 – Transação simples com o banco vazio



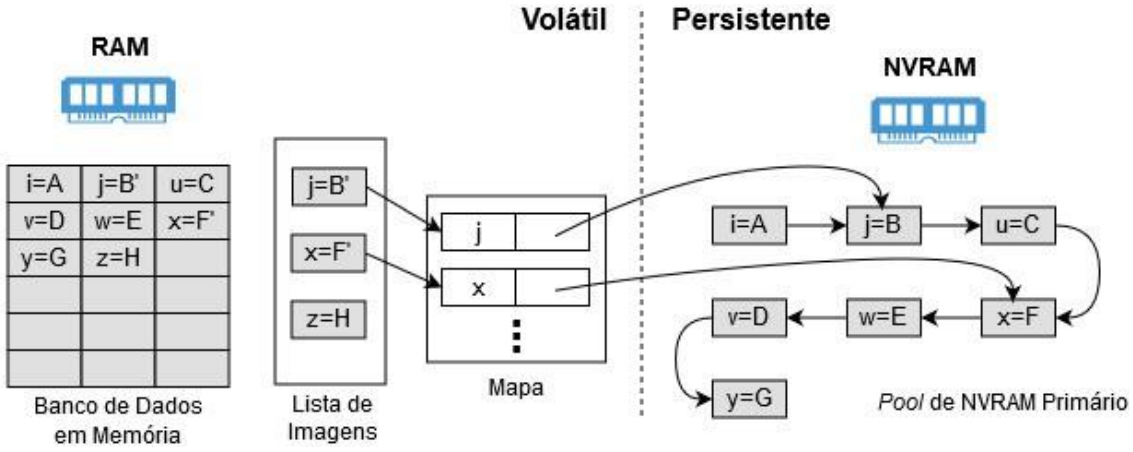
O processo detalhado de confirmação é ilustrado na Figura 11. Ao entrar na etapa de confirmação de uma transação, o SGBD verificará, para cada registro com imagem presente na lista de imagens, se há ou não uma imagem referente a esse registro presente no *pool* primário de NVRAM. Essa verificação é feita pelo mapa, uma estrutura de dados em memória volátil que mapeia o identificador único do registro à ponteiros para a localização de sua imagem no *pool* primário. Caso não seja encontrada uma entrada de imagem do registro, essa imagem será adicionada ao *pool* primário de NVRAM e o mapa em memória volátil será atualizado com essa nova entrada. Caso seja encontrada uma entrada, então a localização da imagem no *pool* de NVRAM é recuperada por meio do mapa e o seu valor será atualizado no local.

A etapa de confirmação ocorre de maneira serial, ou seja, apenas uma transação poderá realizá-la por vez. Essa característica é importante, pois garantirá que, em nenhum momento, mais de uma *thread* escreverá no *pool* primário ao mesmo tempo, evitando condições de corrida. Esse ponto de contenção é semelhante ao buffer utilizado para armazenar a calda do log nas abordagens baseadas em ARIES no qual apenas uma transação pode adicionar suas suas informações por vez.

É importante ressaltar que ETERNAL não limita concorrência durante a execução das transações em memória RAM, sendo ortogonal à protocolos de controle de concorrência. Isso ocorre porque a etapa de confirmação se inicia apenas quando a transação atinge o estado de parcialmente confirmada (sessão 2.3.6), ou seja, todas as operações de leitura e escrita já

foram realizadas em memória principal e a lista de imagens já se encontra construída. Dessa forma, ETERNAL é indiferente ao controle de concorrência utilizada no momento da execução das operações e também é indiferente à técnica utilizada para realizar o *rollback* de transações canceladas. Embora essa restrição gere uma pequena sobrecarga devido à contenção durante a etapa de confirmação, tal sobrecarga é atenuada pela rapidez com que a lista de imagens é persistida em NVRAM devido a sua baixíssima latência de acesso. A mesma estratégia seria completamente inviável em dispositivos de alta latência como discos e SSDs.

Figura 11 – Etapa de confirmação



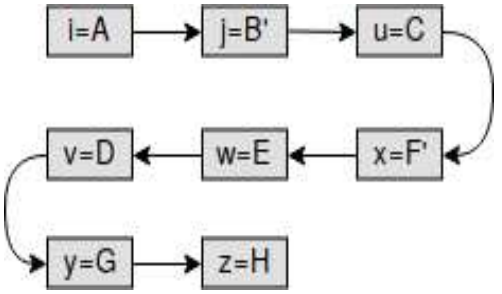
No exemplo da Figura 11, temos uma transação que atualizou o valor do registro j de B para B', atualizou o valor do registro x de F para F' e inseriu uma novo registro z com valor H. A lista de imagens dessa transação, portanto, possui uma imagem para cada registro escrito. Os registros j e x possuem entradas no mapa, indicando que há imagens desses registros no *pool* primário, já z não possui entrada no mapa, indicando que será necessário adicionar uma nova imagem no *pool* primário quando da confirmação da transação.

Observa-se que o *pool* primário de NVRAM é organizado no formato de uma lista encadeada. Essa estrutura de dados foi escolhida para que a inserção de novas entradas seja realizada com o menor custo possível, O(1), sendo o mapa em memória volátil utilizado para indexar a localização das imagens no *pool*. A única situação onde a lista precisará ser percorrida é no momento da recuperação após falha, pois, nesse caso, todos os dados em memória volátil estariam perdidos sendo necessária uma varredura completa durante o processo de recuperação a ser discutido posteriormente.

Na Figura 11, é possível perceber que o banco de dados em memória principal já possui os dados atualizados com as escritas da transação, mas o *pool* de NVRAM primário ainda

possui a imagem dos valores antigos de B e F, além de não possuir uma imagem de H. A etapa de confirmação do exemplo, portanto, consiste em atualizar as imagens dos registros j e x presentes no *pool* de NVRAM primário, bem como adicionar a imagem H referente ao registro z.

Figura 12 – Pool primário após confirmação



O estado do *pool* de NVRAM primário após o término da etapa de confirmação é ilustrado pela Figura 12. Percebe-se que todas as imagens que estavam presentes na lista de imagens da transação foram persistidas no *pool* primário, completando a etapa de confirmação. Pode-se perceber também que, como z foi um novo registro inserido pela transação, sua imagem H foi adicionada ao final da lista encadeada deste *pool*. Todos os passos necessários para a realização da etapa de confirmação estão sumarizados no Algoritmo 1.

Algoritmo 1: Protocolo de Confirmação de uma Transação T

```
   Data: Lista de Imagens da Transação T: LI, Mapa em Memória Principal: M,
         Lista Encadeada Presente no Pool Primário: L1
   Result: Transação T Confirmada
 1 for cada imagem de registro img ∈ LI do
 2     if M.contains(img.rid) then
 3         endereço_nvram = M.locate(img.rid);
 4         endereço_nvram = img;
 5     else
 6         endereço_nvram = alocar_nova_imagem_nvram(img);
             L1.insert(endereço_nvram);
 7         M.insert(<img.rid, endereço_nvram>);
 8     end
 9 end
10
```

Fonte: elaborada pelo autor.

O ETERNAL faz uso da NVRAM como meio de persistência dos dados necessários à etapa de confirmação. Trata-se de uma estratégia vantajosa se comparada às abordagens convencionais de persistência em discos magnéticos, pois significa que o tempo de confirmação das transações estará sujeito à latência de acesso à NVRAM, que é próxima à latência de acesso da memória RAM tradicional.

4.2 Atomicidade das Operações em NVRAM

Uma importante característica que deve ser observada é a garantia da atomicidade de transações em NVRAM. Como discutido anteriormente, é comum que bancos de dados em memória principal não realizem a etapa de *undo*. Isso acontece porque, quando o sistema cai, todos os conteúdos presentes em memória volátil são perdidos, portanto não faz sentido desfazer operações realizadas nesse meio de armazenamento uma vez que elas não atingem armazenamento persistente, assemelhando-se aos protocolos que utilizam a política *no-steal* em bancos de dados tradicionais. Quando uma falha ocorre durante a execução de transações que escrevem dados em NVRAM, no entanto, é possível que operações de transações que não foram confirmadas acabem sendo persistidas. Para lidar com esse problema, ETERNAL utiliza logs de *undo*.

Conforme descrito na sessão 2.2, a memória NVRAM é exposta para ETERNAL como uma região de memória contínua onde bytes individuais podem ser acessados por meio do endereço inicial do bloco somando-se aos *offsets* relativos ao endereço de cada byte. É nessa região de memória que os objetos persistentes são armazenados. No modelo de atomicidade utilizado pela libpmemobj, toda transação deve necessariamente reservar uma região de memória antes de realizar qualquer escrita nela. Uma região de memória é definida pelo *offset* a partir do endereço inicial da NVRAM juntamente com o tamanho da região em bytes. Quando uma transação precisa modificar um certo objeto, a região de memória NVRAM onde ele se encontra é reservada e uma imagem dessa região anterior à modificação é adicinada à um log de *undo*, também em NVRAM. A transação fica livre para modificar diretamente quaisquer objetos nesse intervalo de memória. Em caso de falha ou interrupção, todas as alterações dentro desse intervalo serão revertidas no momento da reinicialização. O log de *undo* é descartado quando a transação é concluída com sucesso.

4.3 Uso Eficiente de NVRAM

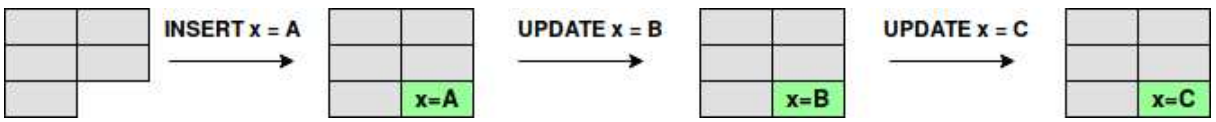
Um aspecto em que ETERNAL é significativamente melhor que as abordagens baseadas em WAL é com relação ao uso das capacidades específicas das memórias do tipo NVRAM, pois nossa abordagem tira proveito da capacidade de endereçamento por byte desse tipo de dispositivo. Em abordagens baseadas em *write-head logging*, novas entradas são concatenadas a calda do log conforme registros vão sendo escritos por transações. Esse tipo de padrão de escrita sequencial é especialmente eficiente em discos e SSDs. A Figura 13 mostra a evolução de um log no caso em que um registro é inserido inicialmente com valor A, em seguida tem o seu valor atualizado para B e finalmente é atualizado para C.

Figura 13 – Abordagem baseada em WAL

No exemplo da Figura 13, um registro passa por três operações de escrita, gerando três entradas em log. Conforme descrito na seção 2.6, o caso onde um mesmo registro é escrito repetidas vezes por diversas transações não é incomum em dados classificados como quentes, de fato esta é uma característica típica de cargas de trabalho OLTP (ZHANG *et al.*, 2016). Nesse tipo de carga, registros acessados mais recentemente têm maiores chances de serem acessados em um futuro próximo, tais registros também são chamados de "quentes". Eventualmente, com o tempo, a taxa de acesso desses registros cai e eles passam a ser chamados de "frios" (ELDAWY *et al.*, 2014).

A Figura 14 mostra um exemplo de como se dá o uso do armazenamento na abordagem ETERNAL. Como as imagens de registros escritos são armazenadas em NVRAM, um meio endereçável por byte, então é possível indexá-las a nível de registro. Quando um mesmo registro é escrito mais de uma vez, basta atualizar a sua última imagem presente no *pool* primário de NVRAM, pois apenas a última imagem é necessária ao protocolo de recuperação.

Figura 14 – Abordagem ETERNAL



O principal aspecto no qual ETERNAL supera as abordagens baseadas em WAL

está relacionado ao comportamento do SGBD no cenário em que o armazenamento NVRAM se esgota. Como demonstrado acima, ETERNAL não precisa manter várias imagens de um mesmo registro que é repetidamente escrito. Essa característica garante que, para uma mesma carga de trabalho OLTP, a situação de esgotamento de NVRAM é menos frequente em ETERNAL do que nas abordagens baseadas em WAL, diminuindo drasticamente a necessidade de recorrer a armazenamentos mais lentos. Além disso, ETERNAL implementa um mecanismo assíncrono de despejo de dados de NVRAM para disco.

4.4 Despejo em Disco

Conforme ilustrado na Figura 8, a arquitetura ETERNAL é composta por dois *pools* de NVRAM. Como já discutido anteriormente, o *pool* de NVRAM primário é utilizado na persistência da lista de imagens durante a etapa de confirmação, fazendo uso da baixa latência de acesso e da capacidade de endereçamento por byte do armazenamento NVRAM como meios para diminuir a sobrecarga da etapa de confirmação.

A inserção de novos dados de imagens no *pool* de NVRAM primário eventualmente levará ao ao esgotamento desse tipo de recurso, não mais possibilitando o uso desse armazenamento. Para sanar esse problema, ETERNAL possui um mecanismo de despejo que visa continuar processando a etapa de confirmação das transações em armazenamento NVRAM ao mesmo tempo que libera esse recurso transferindo seu conteúdo para um outro armazenamento secundário.

A maneira pela qual ETERNAL alcança o objetivo de despejar o conteúdo da NVRAM enquanto a utiliza para processar transações é particionando a memória não volátil em dois *pools* distintos. Enquanto o *pool* primário é utilizado diretamente na etapa de confirmação, o pool secundário é utilizado no processo de despejo.

O processo de despejo é iniciado quando *pool* primário é completamente preenchido por imagens de registros, não podendo mais ser utilizado. Nesse momento ocorre uma rotação de *pools*, ou seja, o *pool* primário, já esgotado, assume o papel do *pool* secundário e o *pool* secundário, normalmente vazio, assume o papel do *pool* primário, passando a receber novas imagens de registros escritos por transações. Uma vez que a rotação é concluída com sucesso, uma *thread* de despejo que atuará sobre o *pool* secundário é disparada.

O papel da *thread* de despejo consiste em percorrer cada uma das imagens de registros que se encontram na lista encadeada presente no *pool* secundário, serializar cada

uma dessas imagens e finalmente as escrever em um arquivo sequencial que se encontra em disco. Uma vez que toda a lista foi percorrida e todo o conteúdo do *pool* secundário se encontra persistido no arquivo sequencial, a *thread* de despejo pode finalmente marcar o *pool* secundário como livre.

A medida que o SGBD processa várias transações, novos despejos continuarão ocorrendo e o arquivo sequencial em disco continuará crescendo. É importante ressaltar que, futuramente, o arquivo será utilizado em um eventual processo se recuperação que será discutido na próxima sessão. Uma possível fonte de problemas é quando uma falha ocorre durante o processo de despejo, isso significa que um despejo ocorreu de maneira parcial, ou seja, algumas imagens foram gravadas no arquivo, mas o *pool* secundário não foi marcado como limpo. Esse despejo incompleto pode ser identificado por meio dos marcadores de despejo. Um marcador de início é escrito no arquivo quando um despejo é iniciado e um marcador de fim é escrito no arquivo após o seu término, sinalizando que o conteúdo do *pool* secundário foi limpo depois de ter todo seu conteúdo despejado para disco.

Todos as passos executados pela *thread* de despejo são ilustrados no Algoritmo 2. A transferência do conteúdo do *pool* secundário para o arquivo de despejo sequencial acontece em um laço nas linhas 2-4, nas linhas 1 e 6 têm-se a escrita dos marcadores de início e de fim de despejo respectivamente. O *pool* é marcado como disponível na linha 5

Algoritmo 2: Protocolo de Despejo

Data: Lista Encadeada Presente no *Pool* Secundário: **L2**, Arquivo de Despejo Sequencial: **ADS**
Result: Despejo do Conteúdo do *Pool* Secundário para o Disco

```
1  ADS.write(marcador_inicio_despejo);
2  for cada imagem de registro img ∈ L2 do
3  |    ADS.write(img);
4  end
5  L2.clear();
6  ADS.write(marcador_fim_despejo);
7
```

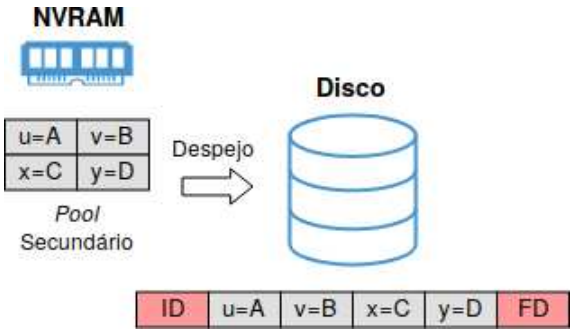
Fonte: elaborada pelo autor.

A Figura 15 ilustra um exemplo de um despejo do *pool* secundário para disco. É possível observar que as imagens A, B, C e D referentes aos registros u, v, x e y respectivamente foram serializadas e escritas no arquivo sequencial. Observe na mesma figura os marcadores de início de despejo (ID) e de fim de despejo (FD) antes e depois das imagens no

arquivo sequencial.

As características da carga de trabalho são cruciais para definir o quão contencioso o processo de despejo será, pois o mecanismo de *pools* rotativos somente trará algum ganho caso o despejo do *pool* secundário seja realizado em *background* enquanto o *pool* primário é utilizado no processo de confirmação. O típico cenário problemático é aquele no qual o *pool* primário é completamente preenchido enquanto o despejo do *pool* secundário ainda está sendo realizado. Nesse caso, novas transações terão que esperar a liberação do *pool* secundário para voltar a realizar novas confirmações. Esse cenário indesejável acontecerá sempre que o tempo de preenchimento do *pool* primário for menor que a duração do despejo do *pool* secundário para disco. Como veremos na análise dos resultados experimentais, esse cenário não ocorre.

Figura 15 – Despejo em Disco



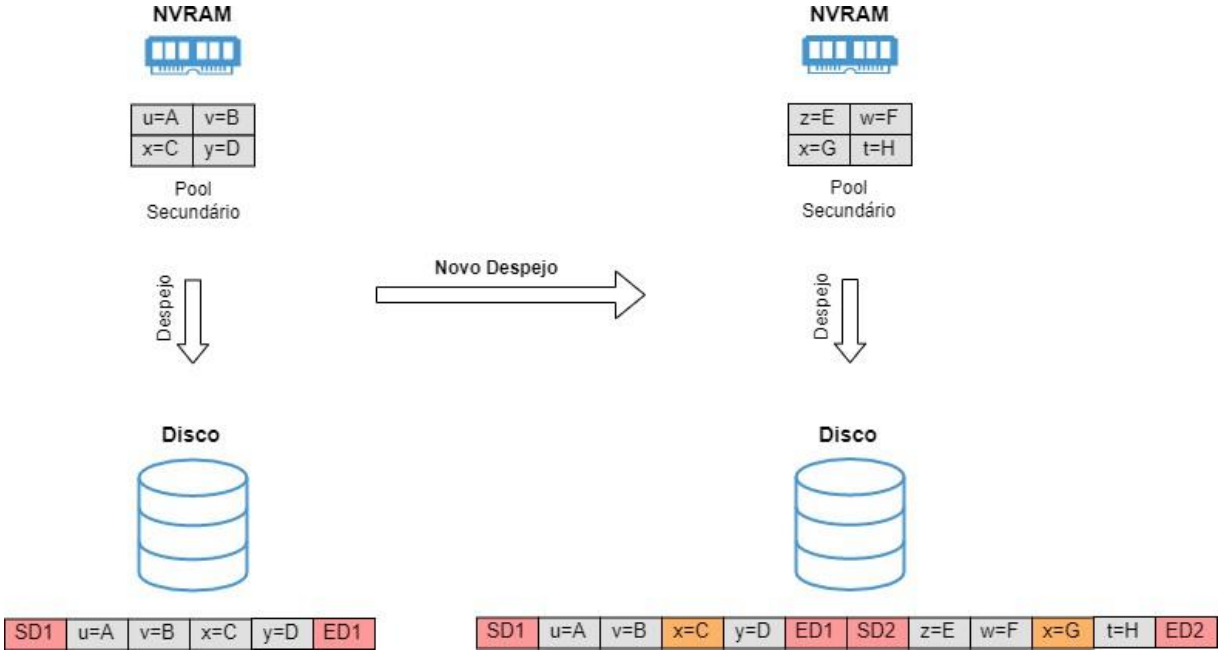
O mecanismo descrito na sessão anterior por meio do qual ETERNAL faz uso eficiente da NVRAM em cargas OLTP evita esse tipo de problema. Ao atualizar apenas a última imagem de um registro, ETERNAL evita a escrita de dados desnecessários, economiza armaze- namento NVRAM e diminui o tempo necessário para preencher o *pool* primário, garantindo uma quantidade maior de tempo do uso do *pool* primário enquanto o processo de despejo pode ser realizado em *background.*

4.5 Protocolo de Recuperação

Quando ocorre uma queda do SGBD, todo o conteúdo presente em memória não volátil é perdido. No caso específico de SGBDs em memória principal, isso significa que todo o banco deverá ser carregado em memória novamente. Em ETERNAL, o protocolo de recuperação funciona carregando a última imagem transacionalmente consistente de cada registro do banco, não sendo necessário um histórico de atualizações como ocorre em abordagens baseadas em *Write-ahead Logging* (WAL).

O protocolo de confirmação garante que toda transação confirmada deve necessariamente persistir as imagens de todos os registros por ela escritos em NVRAM. O protocolo de despejo garante que, conforme o *pool* primário é preenchido, imagens de registros serão eventualmente levadas para o *pool* secundário e finalmente serão persistidas no arquivo sequen- cial em disco. Existem, portanto, três localizações possíveis para as imagens de registros: o *pool* de NVRAM primário, o *pool* de NVRAM secundário e o arquivo sequencial em disco. Resumidamente, o processo de recuperação consiste em percorrer todos essas localizações, identificar a imagem mais atual de cada registro e carregá-la novamente ao banco.

Figura 16 – Múltiplas versões



O protocolo de recuperação deve levar em conta que é possível existir mais de uma versão de imagem de um mesmo registro nos diferentes meios de armazenamento. A figura 16 ilustra um exemplo. À esquerda tem-se a ilustração de um despejo que transfere as imagens dos registros u, v, x e y para o arquivo sequencial, além dos marcadores de início de fim. Logo após esse despejo, o pool secundário será limpo e estará pronto para uma nova rotação como foi descrito na sessão anterior. Conforme novos registros vão sendo inseridos ou atualizados, novos despejos serão realizados.

Um novo despejo é ilustrado à direita da Figura 16. Percebe-se que uma nova imagem do registro x se encontra no pool secundário. Essa nova imagem será persistida no arquivo sequencial como todas as outras. Esse tipo de situação acontecerá sempre que um

registro que já foi despejado para disco for atualizado novamente. As duas entradas de x com imagens C e G encontram-se destacadas no arquivo sequencial gerado pelo novo despejo, mas a última imagem transacionalmente consistente é a x=G.

Além de múltiplas imagens de um mesmo registro no arquivo sequencial, também é possível que uma imagem de um mesmo registro esteja presente no *pool* primário e no *pool* secundário ao mesmo tempo, porém o mapa em memória principal utilizado no protocolo de confirmação assegura que dentro de cada pool haja apenas uma imagem de um mesmo registro.

A versão mais nova da imagem de um registro estará sempre no *pool* de NVRAM primário, pois é nele onde as imagens são inicialmente persistidas durante a etapa de confirmação. Em seguida, temos o *pool* secundário, com versões das imagens presentes no momento do despejo. Por último, temos as imagens presentes no arquivo sequencial, já despejadas em disco, com imagens mais antigas no início do arquivo e mais novas no fim.

Para garantir que a versão final dos registros após a recuperação será a mais nova, o algoritmo de recuperação inicia o carregamento do banco a partir das imagens mais antigas, as sobrescrevendo sempre que encontrar uma imagem mais nova de um registro já carregado. Como é sabido, os marcadores de início e de fim de despejo são utilizados para garantir a atomicidade do despejo em caso de falha. Se, no momento da recuperação, for detectado um despejo com marcador de início, mas sem um marcador de fim, então significa que esse despejo não foi concluído completamente e que esse despejo deve ser ignorado.

O Algoritmo 3 descreve com mais detalhes o processo de recuperação. Ele é composto por três laços, o primeiro entre as linhas 1-7, o segundo entre as linhas 8-10 e o terceiro entre as linhas 11-13. O laço entre as linhas 1-7 é o primeiro a ser executado e representa a varredura o arquivo sequencial em disco. Cada um dos despejos é percorrido, verificando a presença dos marcadores de início e de fim. Os registros cujo despejo possua ambos os marcadores serão inseridos ao banco. Como todos os despejos possuem o mesmo tamanho, que é o tamanho do *pool* secundário, então a localização do marcador de fim de despejo pode ser calculada a partir da localização do marcado de início de despejo. Dessa forma ambos os marcadores podem ser facilmente identificados para cada despejo

Algoritmo 3: Protocolo de Recuperação

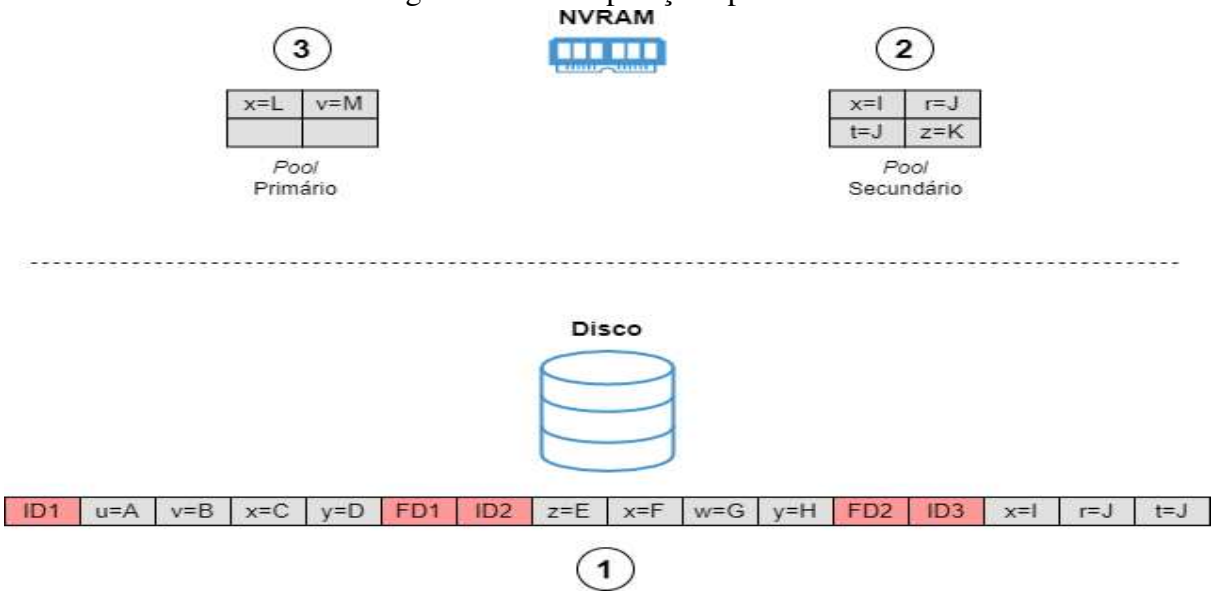
Data: Lista Encadeada Presente no *Pool* Primário: **L1**, Lista Encadeada Presente no *Pool* Secundário: **L2**, Arquivo de Despejo Sequencial: **ADS**, Banco de Dados Vazio em Memória Principal: **BD**
Result: Banco Recuperado ao Seu Último Estado Trasacionalemente Consistente

```
1  for cada despejo d ∈ ADS do
2      if d possui marcador de início e possui marcador de fim then
3          for cada imagem de registro img ∈ d do
4              BD.insert(img);
5          end
6      end
7  end
8  for cada imagem de registro img ∈ L2 do
9      BD.insert(img);
10 end
11 for cada imagem de registro img ∈ L1 do
12     BD.insert(img);
13 end
14
```

Fonte: elaborada pelo autor.

O laço das linhas 8-10 representa a varredura do *pool* secundário em NVRAM, inserindo todos os registros lá presentes. Da mesma forma, as linhas 11-13 a varredura do *pool* primário, também inserindo os registros. A ordem de execução dos laços garante que as imagens mais novas de um dado registro sejam sempre inseridas por último, assegurando a recuperação do último estado transacionalmente consistente.

Figura 17 – Recuperação após falha



A Figura 17 ilustra uma instância de ETERNAL logo após uma falha. O arquivo sequencial em disco é mostrado em 1, o *pool* secundário é mostrado em 2 e o *pool* primário é mostrado em 3. Percebe-se que, neste momento, não há banco materializado em memória principal, pois todo o conteúdo em RAM foi perdido devido à falha.

No exemplo da Figura 17, o processo de recuperação se inicia com a varredura do arquivo sequencial em disco em 1. O primeiro despejo em disco possui ambos os marcadores ID1 e FD2, portanto todos as imagens nele presentes (u, v, x, e y) serão inseridas à nova instância do banco em memória. Neste primeiro momento, o estado do banco ficaria {u=A, v=B, x=C, y=D}.

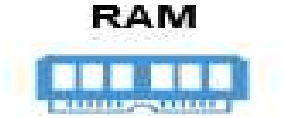
O próximo despejo também possui os marcadores de início e de fim de despejo ID2 e FD2 respectivamente, portanto os suas imagens também serão inseridas ao banco. Percebe-se que além das imagens de z e w, tem-se novas imagens de x e y, que já haviam sido anteriormente carregadas ao banco. Nesse caso as imagens atuais sobrescreverão as imagens carregadas anteriormente, deixando o estado do banco como {z=E, w=G, u=A, v=B, x=F, y=H}

O próximo despejo presente no arquivo sequencial em disco possui um marcador de início ID3, porém não possui um marcador de fim. Como explicado anteriormente, isso significa que o despejo não chegou a ser concluído. Nesse caso todos os registros desse despejo serão ignorados, pois ainda se encontram no *pool* secundário. O estado do banco continua como {z=E, w=G, u=A, v=B, x=F, y=H}.

Em seguida ocorrerá a varredura do *pool* secundário mostrado em 2. As imagens dos registros r e t são novas e serão inseridas no banco, já as imagens de x e z serão apenas atualizadas. O estado do banco após a recuperação do conteúdo do *pool* secundário seria {r=J, t=J, z=K, w=G, u=A, v=B, x=I, y=H}.

Finalmente ocorrerá a varredura do *pool* primário. Os registros presentes são v, que será inserido ao banco pela primeira vez, e x, que mais uma vez terá a sua imagem atualizada. Portanto o estado final do banco materializado em memória principal ficará como {v=M, r=J, t=J, z=K, w=G, u=A, v=B, x=L, y=H}, conforme ilustrado pela Figura 18.

Figura 18 – Banco após a recuperação



x=L	v=M
r=J	t=J
z=K	w=G
y=H	u=A

Este capítulo descreveu os componentes e o funcionamento da arquitetura ETERNAL. Foram apresentados em detalhes como funcionam os protocolos de confirmação e de despejo, bem como o funcionamento do algoritmo de recuperação. Foram discutidas também como ETERNAL faz uso das capacidades únicas de endereçamento por byte para atualizar as imagens de registros *in place*, evitando o uso desnecessário de NVRAM e permitindo o despejo do seu conteúdo para um armazenamento tradicional em disco sem a interrupção do processamento de novas transações.

5 RESULTADOS

Apresenta-se agora a análise experimental comparando o desempenho de ETERNAL com a abordagem estado da arte baseada em WAL e utilizada em (MALVIYA *et al.*, 2014) para banco de dados em memória principal. Para tornar a comparação mais justa, a abordagem concorrente foi adaptada para utilizar NVRAM como meio de armazenamento para seu log. Tal como ocorre em ETERNAL, o log da abordagem WAL será despejado para disco caso o armazenamento NVRAM seja completamente preenchido. Chamaremos essa abordagem de NVWAL.

5.1 Configurações do Experimento

Os experimentos foram realizados em uma máquina Intel Core i7-7800X com 6 núcleos e 12 threads rodando a uma frequência base de 3.5GHz e 128GB de memória SDRAM DDR4, com sistema operacional Ubuntu 18.04. As configurações encontram-se sumarizadas na Tabela 4.

Tabela 4 – Configurações do Experimento

Componente	Configuração
Processador	Core i7-7800X
Núcleos/*Threads*	6/12
Cache	8.25MB L3
RAM	128GB DDR4
NVRAM	2GB DDR4 (Emulada)
Disco	WD Blue SATA 3 2TB 7200RPM
Sistema Operacional	Ubuntu 18.04 LTS

Fonte: elaborada pelo autor.

A emulação de NVRAM é baseada na memória RAM que será vista pelo sistema operacional como uma região de memória persistente. Por ser uma emulação baseada em RAM, é provável que seja um pouco mais rápida que uma memória não volátil, mas isso não deve ser comprometer o resultado dos experimentos, pois ambas em ambas as abordagens a serem comparadas utilizarão o mesmo tipo de emulação.

Assim como acontece em trabalhos que se diferenciam fundamentalmente da maneira como SGBDs tradicionalmente funcionam, ETERNAL e NVWAL foram implementados como subsistemas *standalone* em C++ (KIMURA, 2015) (ZHENG *et al.*, 2014). Para fins de avaliação, ambos os subsistemas de recuperação foram conectados a um banco de dados em memória

simples baseado na coleção std::unordered_map da biblioteca padrão de C++. Tanto o banco quanto as *engines* foram compilados em um mesmo binário junto com uma implementação do *benchmark* YSCB em C++.

5.2 Carga de Trabalho

O *benchmark* usado no experimento é o YCSB (COOPER *et al.*, 2010), utilizado para modelar aplicações OLTP. O YCSB disponibiliza por padrão cinco cargas de trabalho. Todas as cargas encontram-se ilustradas na Tabela 6. Pode ser visto que, na coluna Seleção de Registros, encontram-se as distribuições Zipfian, Latest e Uniforme. Além disso, tem-se também as proporções de operações mostradas na coluna Operações. Juntas, as distribuição e as proporções de operações modelam as cargas de trabalho padrão do YCSB. Exemplos de aplicações reais que podem ser modeladas por essas cargas são mostrados na coluna Exemplo de Aplicação.

Tabela 5 – Cargas de trabalho padrão do YCSB

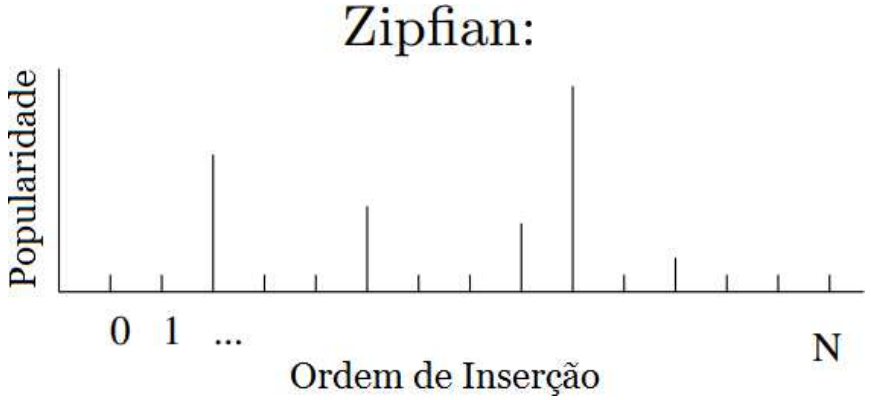
Carga de Trabalho	Operações	Seleção de Registros	Exemplo de Aplicação
Carga A - Pesada em Atualizações	Leitura: 50% Atualização: 50%	Zipfian	Armazenamento de sessão, gravando ações recentes em uma sessão do usuário.
Carga B - Pesada em Leituras	Leitura: 95% Atualização: 5%	Zipfian	Marcação de fotos; adicionar uma marcação é uma atualização, mas a maioria das operações é para ler marcações.
Carga C - Apenas Leituras	Leitura: 100% Atualização: 0%	Zipfian	*Cache* de perfil de usuário, onde os perfis são construídos em outro local.
Carga D - Ler Últimos	Leitura: 95% Inserção: 5%	Latest	Atualizações de *status* de usuário, pessoas querem ler os últimos *status.*
Carga E - Intervalos curtos	Varredura: 95% Inserção: 5%	Zipfian/Uniforme	Conversação por *threads*, onde cada varredura é para as postagens em uma *thread.*

Fonte: elaborada pelo autor.

Na Carga E, a distribuição Uniforme é utilizada apenas na definição do tamanho do um intervalo a ser percorrido para leitura, porém o acesso ao primeiro registro do intervalo é gerado a partir da distribuição Zipfian. Apenas as distribuições Zipfian e Latest são utilizadas para gerar padrões de acesso de escrita a registros, portanto apenas essas distribuições serão utilizadas nos experimentos, já que cargas de leitura não possuem qualquer impacto sobre o subsistema de recuperação.

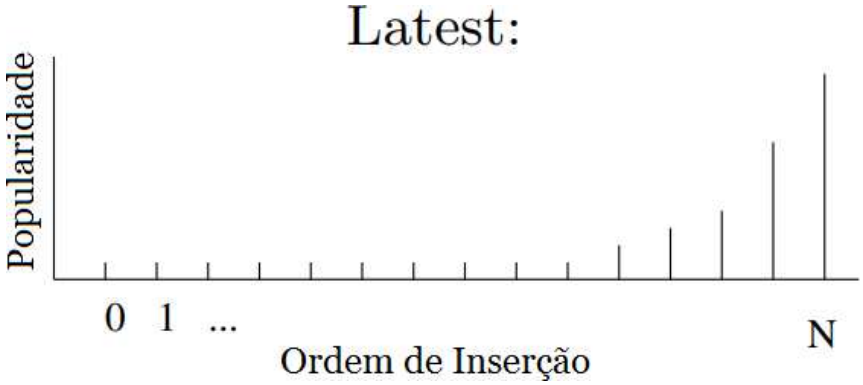
Tanto a distribuição Zipfian quanto a Latest possuem o padrão de acesso *skewed*. Isso significa que, a cada intervalo de funcionamento do banco, a distribuição de acesso à alguns dados é privilegiada sobre outros. Esse padrão de acesso foi descrito na sessão 4.3. Analisando as Figuras 19 e 20 temos dois histogramas que descrevem as distribuições Zipfian e Latest respectivamente. No eixo x tem-se listado cada um dos N registros presentes no banco um dado momento, onde 0 representa o primeiro elemento ser inserido e N representa o último. No eixo y tem-se a popularidade de cada registro, ou seja, a sua probabilidade de receber um acesso.

Figura 19 – Distribuição zipfian



Comparando as Figuras 19 e 20 é possível perceber as diferenças entre as distribuições Zipfian e Latest. Sua maior diferença é o padrão de acesso aos registros de acordo com a sua ordem de inserção. Com a distribuição Zipfian, itens considerados populares mantém a sua popularidade mesmo após a inserção de novos itens. Já com a distribuição Latest, a popularidade de acesso muda a cada nova inserção, onde o registro recentemente inserido é considerado o mais o mais popular.

Figura 20 – Distribuição latest



A fim de avaliar ETERNAL e NVWAL, dois tipos de configurações de operações de escrita foram definidas no experimento:

- **Atualização.** 100% são das operações são atualizações.
- **Inserção.** 50% das operações são inserções e 50% são atualizações.

Cada configuração de operações foi ainda combinada com as distirbuições Zipfian e Lastest, resultando em um total de quatro cargas a serem utilizadas nos experimentos. As características das cargas podem ser vistas na Tabela 6.

Tabela 6 – Cargas de trabalho padrão do experimento

Carga	Operações	Distribuição
Atualização com distribuição Zipfian	Atualização: 100%	Zipfian
Atualização com distribuição Latest	Atualização: 100%	Latest
Inserção com distribuição Zipfian	Inserção: 50% Atualização: 50%	Zipfian
Inserção com distribuição Latest	Inserção: 50% Atualização: 50%	Latest

Fonte: elaborada pelo autor.

Antes da execução de cada carga, banco foi inicializado com 12GB de registros.

5.2.1 Resultados Experimentais

Nas Figuras 21 e 22 são apresentados os gráficos de *throughput* que mostram o número de transações realizadas a cada segundo na carga de atualização utilizando as distribuições zipfian e latest respectivamente. Temos também duas linhas representando o *throughput* médio total do experimento para as duas abordagens. ETERNAL apresentou um *throughput* médio de 174327 TXs/s contra 99792 TXs/s de NVWAL com a distribuição

zipfian. Com a distribuição lastest, ETERNAL apresentou um *throughput* médio de 167650 TXs/s contra 97939 TXs/s de NVWAL.

Figura 21 – *Throughput*: atualização com distribuição zipfian

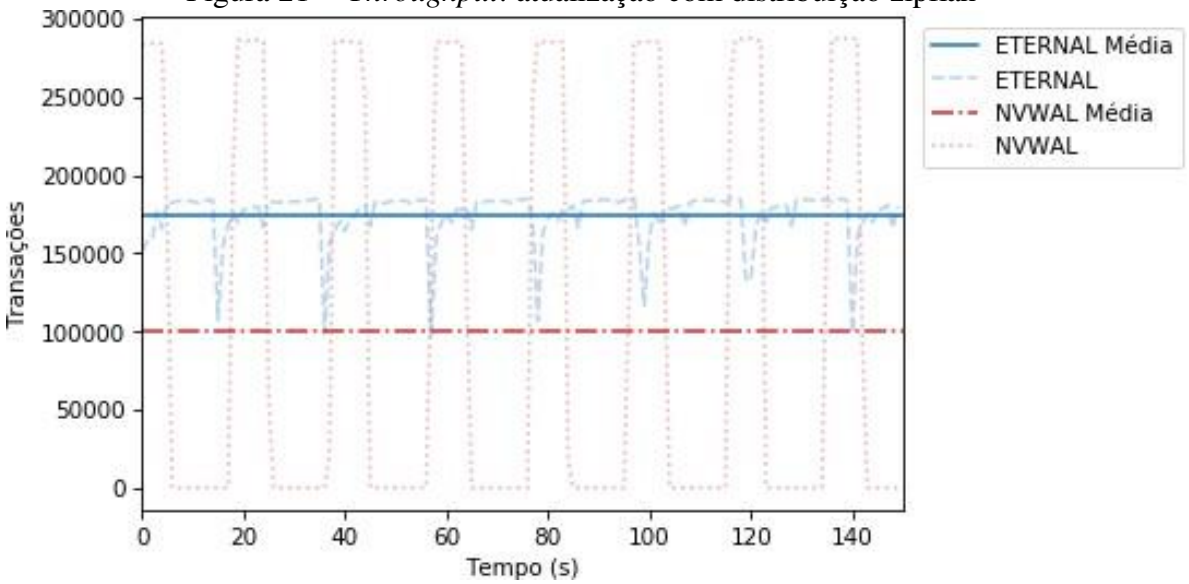
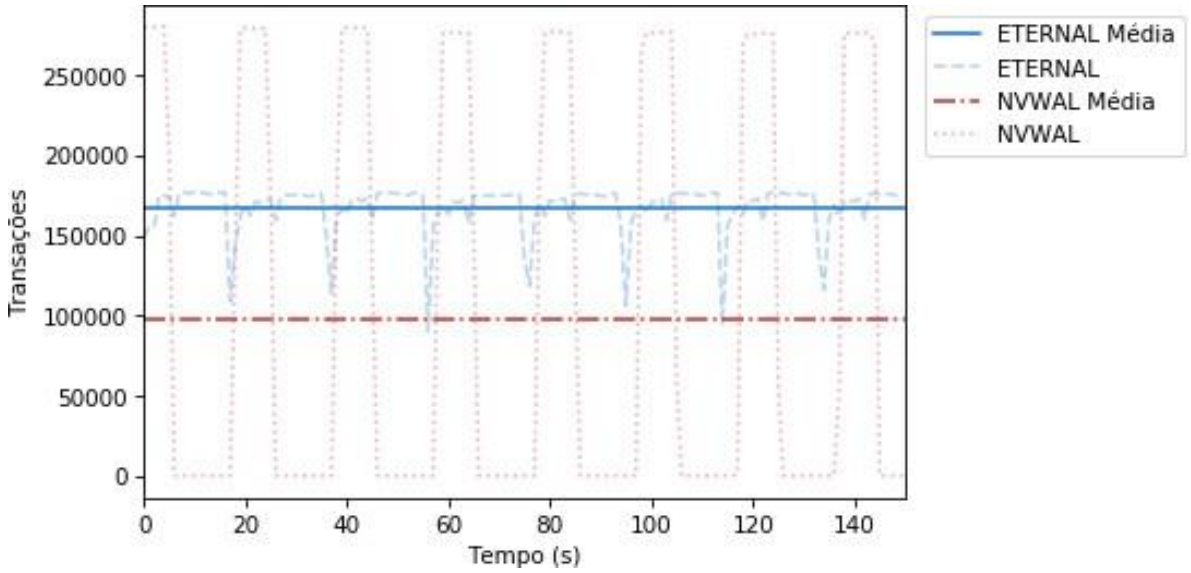


Figura 22 – *Throughput*: atualização com distribuição latest



Nas Figuras 23 e 24 são apresentados os gráficos de *throughput* que mostram o número de transações realizadas a cada segundo na carga de inserção utilizando as distribuições zipfian e latest respectivamente. ETERNAL apresentou um *throughput* médio de 106000 TXs/s contra 83579 TXs/s de NVWAL com a distribuição zipfian. Com a distribuição lastest, ETERNAL apresentou um *throughput* médio de 104495 TXs/s contra 82438 TXs/s de NVWAL.

Figura 23 – *Throughput*: inserção com distribuição zipfian

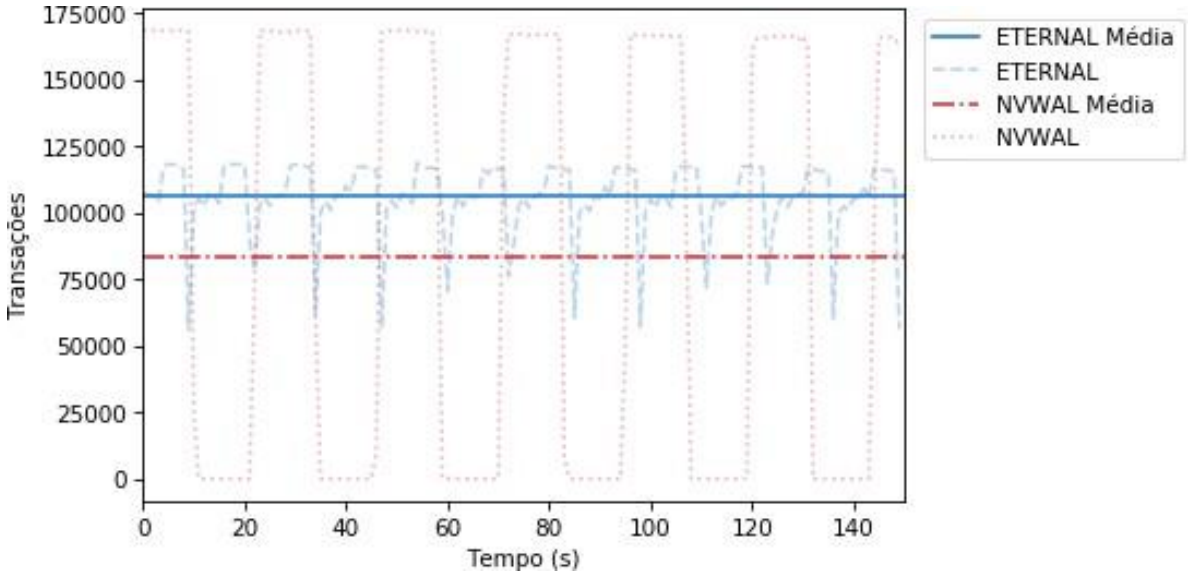
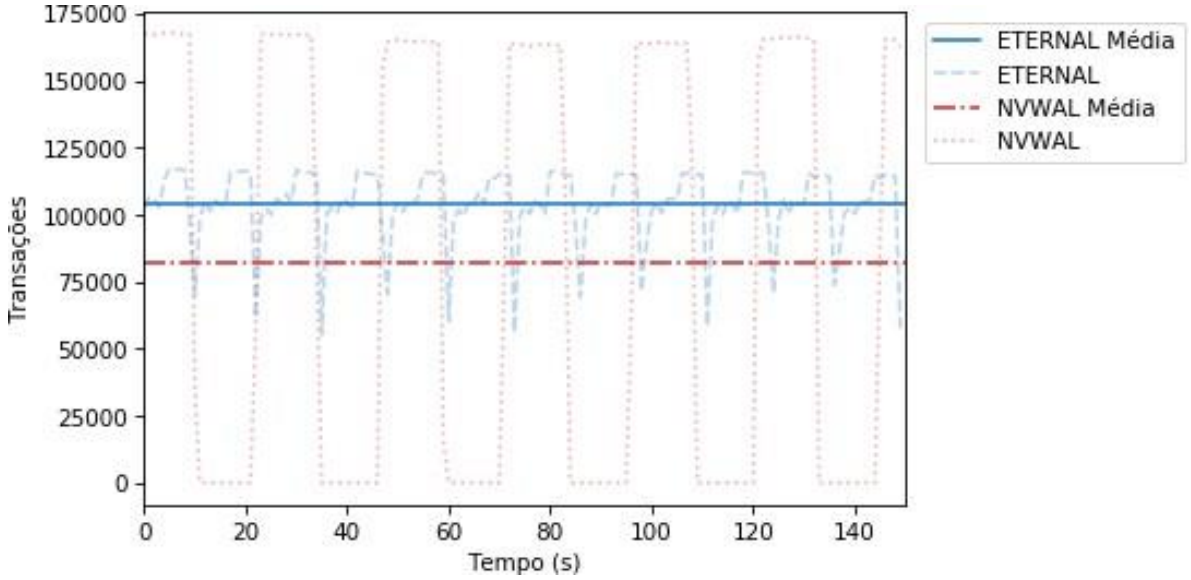


Figura 24 – *Throughput*: inserção com distribuição latest



Os resultados dos experimentos com cada uma das cargas avaliadas foram sumarizados abaixo na Tabela 7.

Tabela 7 – Resultados experimentais

Carga	*Throughput* Médio NVWAL (TXs/s)	*Throughput* Médio ETERNAL (TXs/s)	Diferença (%)
Atualização com distribuição Zipfian	99792	174327	75
Atualização com distribuição Latest	97939	167650	71
Inserção com distribuição Zipfian	83579	106000	27
Inserção com distribuição Latest	82438	104495	27

Fonte: elaborada pelo autor.

5.2.2 Análise dos Resultados Experimentais

Analisando os resultados, é possível perceber que, na média, ETERNAL atinge *throughputs* significativamente maiores que NVWAL. Pelos gráficos é possível perceber também que NVWAL apresenta, em alguns momentos, valores de *throughput* maiores que os de ETER- NAL. Essa perda momentânea de desempenho pode ser explicada em parte pela sobrecarga adicional causada pela uso do mapa para acessar as imagens no *pool* primário de NVRAM.

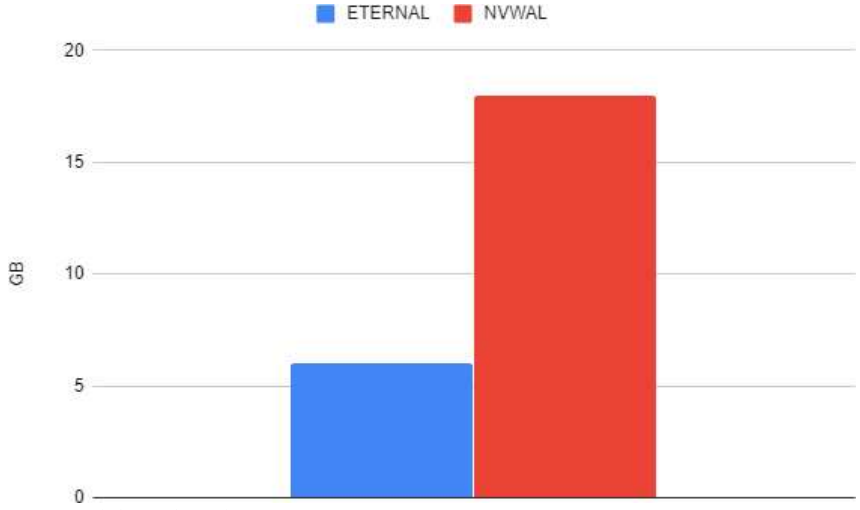
Outro fator a ser considerado é a sobrecarga gerada pelos logs de *undo* da biblioteca libpmemobj, necessários para garantir a atomicidade das operações realizadas em NVRAM.

Foi verificado que o ganho de ETERNAL foi maior nas cargas de atualização que nas cargas de inserção. A explicação para esse fato é o mecanismo por meio do qual ETERNAL sobrescreve a última imagem de um dado no *pool* primário. Se um mesmo dado é repetidamente atualizado, nenhum espaço adicional se faz necessário no *pool* primário, portanto é esperada que uma frequência menor de despejos ao disco. Na carga de inserção, no entanto, quando um novo registro é inserido ao banco, ele deve necessariamente consumir um espaço adicional no *pool* primário, aumentando a frequência de despejos e consequentemente diminuindo o desempenho.

Foi verificado também que os resultados relativos às cargas de atualização e inserção não variaram significativamente com relação à distribuição escolhida. ETERNAL consegue se aproveitar do fato que, em ambas as distribuições, o acesso aos dados é *skewed*, podendo fazer uso do seu mecanismo de sobrescrita de uma mesma imagem em ambos os casos, o que explica os resultados semelhantes. Já NVWAL independe da distribuição escolhida, pois toda escrita a um dado gera necessariamente uma entrada de log.

A vantagem de ETERNAL sobre NVWAL se dá no momento em que a NVRAM se esgota, pois, ao fazer o uso eficiente da tecnologia, a abordagem conseguiu diminuir a quantidade de dados despejados para disco. Por causa desse mecanismo, o *pool* se esgota com menor frequência, permitindo um despejo sem contenção realizado por uma *thread* de fundo. NVWAL despeja, de maneira síncrona, uma quantidade muito maior de dados, o que explica as quedas de *throughput* apresentadas nos gráficos da abordagem.

Figura 25 – Comparação da quantidade de acesso a disco



Fonte: elaborada pelo autor.

Caso NVWAL utilizasse *pools* de NVRAM, de nada adiantaria, pois ambos os pools seriam preenchidos rapidamente, ocasionando bloqueios de espera pelo despejo. Uma forma de demonstrar a quantidade de acesso a disco evitada por ETERNAL é comparar o tamanho do arquivo sequencial despejado em disco com o tamanho do log gerado por NVWAL. Ao final do experimento, com a carga de atualização, o arquivo sequencial de ETERNAL possuía um tamanho de 6GB, já o log gerado por NVWAL possuía 18GB, como pode ser observado na Figura 25. Portanto, NVWAL realizou 3 vezes mais acessos a disco que ETERNAL, o que também explica o seu desempenho geral inferior.

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 Conclusões

A ideia principal por trás de ETERNAL é que o mecanismo por meio do qual SGBDs tradicionais garantem a atomicidade e a durabilidade de suas transações pode ser significativamente aprimorado ao se utilizar as capacidades únicas de armazenamentos do tipo NVRAM. Ao invés de inserir custosas novas entradas em log para cada atualização realizada no banco, ETERNAL tira proveito da baixa latência de acesso e da eficiência do acesso aleatório da NVRAM para persistir e manter apenas a última imagem necessária à recuperação do banco em caso de falha. O trabalho propôs ainda um mecanismo que leva em consideração o *skew* das cargas de trabalho OLTP para a realização do despejo assíncrono dos dados de recuperação presentes em NVRAM. Esse mecanismo permite que as transferências de dados de NVRAM para disco fiquem fora do caminho crítico da etapa de confirmação das transações.

Os experimentos mostraram que a abordagem ETERNAL foi bem sucedida em sua proposta. Em todos os experimentos, ETERNAL apresentou um *throughput* significativamente maior que a abordagem baseada em WAL. Os experimentos mostraram ainda que, em ETERNAL, em nenhum momento as transações tiveram que ser suspensas para a realização de despejos para disco, demonstrando que o mecanismo proposto de despejo assíncrono funcionou mesmo em cenários desfavoráveis como aquele com elevado percentual de inserções. Os experimentos mostraram também que ETERNAL realizou significativamente menos escritas em disco que a abordagem baseada em WAL.

Uma limitação que deve ser considerada em ETERNAl está relacionada à certas cargas de trabalho como as do tipo *bulk loading*. Essas cargas apresentam apenas operações de inserção, o que significa que o mecanismo pelo qual ETERNAL sobrescreve imagens de registros anteriormente atualizados não pode ser utilizado, já que registros inseridos são necessariamente novos. Nesse caso, ETERNAL se comportará semelhante às abordagens baseadas em WAL, inserindo novas entradas em NVRAM para cada nova operação e bloqueando novas transações durante o despejo do conteúdo da NVRAM para disco.

6.2 Trabalhos Futuros

A partir de ETERNAL, muitos trabalhos futuros podem ser realizados. O *pool* secundário é completamente esvaziado durante a realização de um despejo. É possível melhorar

esse mecanismo por meio da classificação dos dados no *pool* de NVRAM em quentes e frios, permitindo que imagens de registros mais acessados permaneçam no *pool*, evitando o custo de adicionar novas imagens do mesmo registro nas próximas rotações. Outro trabalho a ser realizado é a implementação de uma recuperação em tempo real, na qual o banco pode ser colocado em funcionamento imediatamente após a falha. Essa estratégia pode ser particularmente eficaz em ETERNAL, já que as imagens relativas aos registros mais acessados já se encontram em armazenamento NVRAM de rápido acesso. Outra melhoria pode ainda ser realizada para mitigar a limitação relacionada à cargas do tipo *bulk loading* descrita na seção 6.1. Um mecanismo adaptativo pode ser implementado de tal maneira que possa haver uma mudança da estratégia adotada por ETERNAL para uma estratégia de log tradicional caso seja detectado que uma carga do tipo *bulk loading* está sendo executada. O trabalho pode ainda ser estendido com a realização de novos experimentos em outros *benchmarks* OLTP como o TPC-C.

REFERÊNCIAS

ABADI, D. J.; MADDEN, S.; HACHEM, N. Column-stores vs. row-stores: how different are they really? In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 8., 2008, Vancouver. **Proceedings...** Vancouver: [s.n.], 2008. p. 967-980. Disponível em: http://doi.acm.org/10.1145/1376616.1376712. Acesso em: 12 ago. 2019.

AMORA, P. R. P.; TEIXEIRA, E. M.; PRACIANO, F. D. B. S.; MACHADO, J. C. Smartltm: Smart larger-than-memory storage for hybrid database systems. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 33., 2018, Rio de Janeiro (RJ). **Anais...** Rio de Janeiro: [s.n.], 2018. p. 13-24. Disponível em: http: //sbbd.org.br/2018/wp-content/uploads/sites/5/2018/08/013-sbbd_2018-fp.pdf. Acesso em: 17 ago. 2019.

ARULRAJ, J.; PAVLO, A. How to build a non-volatile memory database management system. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 17., 2017, Chicago. **Proceedings...** Chicago: [s.n.], 2017. p. 1753-1758. Disponível em: https://doi.org/10.1145/3035918.3054780. Acesso em: 12 ago. 2019.

ARULRAJ, J.; PAVLO, A.; MENON, P. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 16., 2016, San Francisco. **Proceedings...** San Francisco: [s.n.], 2016. p. 583-598. Disponível em: http://doi.acm.org/10.1145/2882903.2915231. Acesso em: 20 ago. 2019.

ARULRAJ, J.; PERRON, M.; PAVLO, A. Write-behind logging. **VLDB Endowment**, v. 10, n. 4, p. 337-348, 2016. Disponível em: http://www.vldb.org/pvldb/vol10/p337-arulraj.pdf. Acesso em: 15 set. 2019.

COOPER, B. F.; SILBERSTEIN, A.; TAM, E.; RAMAKRISHNAN, R.; SEARS, R. Benchmarking cloud serving systems with YCSB. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 10., 2010, Indianapolis. **Proceedings...** Indianapolis: [s.n.], 2010. p. 143-154. Disponível em: https://doi.org/10.1145/1807128.1807152. Acesso em: 12 ago. 2019.

COURTLAND, Rachel. **Spin Memory Shows Its Might**. [S.l.]: Spectrum, 2014. Disponível em: https://spectrum.ieee.org/. Acesso em: 03 set. 2018.

DEBRABANT, J.; PAVLO, A.; TU, S.; STONEBRAKER, M.; ZDONIK, S. Anti-caching: A new approach to database management system architecture. **VLDB Endowment**, v. 6, n. 14, p. 1942–1953, 2013.

ELDAWY, A.; LEVANDOSKI, J.; LARSON, P.-Å. Trekking through siberia: Managing cold data in a memory-optimized database. **VLDB Endowment**, v. 7, n. 11, p. 931–942, 2014.

FAERBER, F. *et al.* Main memory database systems. Foundations and Trends® in Databases, **Now Publishers Inc.**, v. 8, n. 1-2, p. 1-130, 2017.

FUNKE, F.; KEMPER, A.; NEUMANN, T. Compacting transactional data in hybrid OLTP & OLAP databases. **VLDB Endowment**, v. 5, n. 11, p. 1424–1435, 2012.

GARCIA-MOLINA, H.; SALEM, K. Main memory database systems: An overview. **IEEE Transactions on Knowledge and Data Engineering**, v. 4, n. 6, p. 509-516, 1992.

GOMES, D. B.; BRAYNER, A. R. A.; MACHADO, J. C. Eternal: Uma estratégia eficiente de tolerância a falhas utilizando memória não-volátil. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 34., 2019, Fortaleza (CE). **Anais...** Fortaleza: [s.n.], 2019. p. 21-25.

GRAEFE, G.; KIMURA, H.; KUNO, H. Foster b-trees. **ACM Transactions on Database Systems (TODS)**, v. 37, n. 3, p. 17, 2012.

GRAY, J. N. Notes on data base operating systems. In: GRAY, J. N. **Operating Systems**. [S.l.]: Springer, 1978.

HAERDER, T.; REUTER, A. Principles of transaction-oriented database recovery. **ACM Computing Surveys (CSUR)**, v. 15, n. 4, p. 287-317, 1983.

HARIZOPOULOS, S.; ABADI, D. J.; MADDEN, S.; STONEBRAKER, M. Oltp through the looking glass, and what we found there. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 8., 2008, Vancouver. **Proceedings...** Vancouver: [s.n.], 2008. p. 981-992.

HUANG, J.; SCHWAN, K.; QURESHI, M. K. Nvram-aware logging in transaction systems. **VLDB Endowment**, v. 8, n. 4, p. 389-400, 2014.

KERRISK, M. **The Linux programming interface:** a Linux and UNIX system programming handbook. [S.l.]: No Starch Press, 2010.

KIMURA, H. Foedus: Oltp engine for a thousand cores and nvram. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 15., 2015, San Francisco. **Proceedings...** San Francisco: [s.n.], 2015. p. 691-706.

LANG, H. *et al*. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 16., 2016, San Francisco. **Proceedings...** San Francisco: [s.n.], 2016. p. 311-326. Disponível em: http://doi.acm.org/10.1145/2882903.2915231. Acesso em: 20 ago. 2019.

LEE, B. C.; IPEK, E.; MUTLU, O.; BURGER, D. Phase change memory architecture and the quest for scalability. **Communications of the ACM**, v. 53, n. 7, p. 99-106, 2010.

LEVANDOSKI, J. J.; LARSON, P.-Å.; STOICA, R. Identifying hot and cold data in main-memory databases. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 29., 2013, [S.l.]. **Proceedings...** [S.l.: s.n.], 2013. p. 26-37.

LEVANDOSKI, J. J.; LOMET, D. B.; SENGUPTA, S. **The bw-tree:** A b-tree for new hardware platforms. [S.l.]: IEEE Computer Society, 2013. p. 302-313.

LORIE, R. A. Physical integrity in a large segmented database. **ACM Transactions on Database Systems (TODS)**, v. 2, n. 1, p. 91-104, 1977.

MALVIYA, N.; WEISBERG, A.; MADDEN, S.; STONEBRAKER, M. **Rethinking main memory OLTP recovery.** [S.l.]: IEEE Computer Society, 2014. p. 604–615.

MAO, Y.; KOHLER, E.; MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 7., 2012, [S.l.]. **Proceedings...** [S.l.: s.n.], 2012. p. 183-196.

MOHAN, C.; HADERLE, D.; LINDSAY, B.; PIRAHESH, H.; SCHWARZ, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. **ACM Transactions on Database Systems (TODS)**, v. 17, n. 1, p. 94-162, 1992.

MOHAN, C.; HADERLE, D. J.; LINDSAY, B. G.; PIRAHESH, H.; SCHWARZ, P. M. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. **ACM Transactions on Database Systems**, v. 17, n. 1, p. 94-162, 1992.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems.** 3. ed. [S.l.]: McGraw-Hill, 2003.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts, Seventh Edition**. [S.l.]: McGraw-Hill Book Company, 2020. Disponível em: https://www.db-book.com/db7/index.html. Acesso em: 29 jun. 2019.

STORAGE AND NETWORKING INDUSTRY ASSOCIATION - SNIA. **Nvm programming model (npm) v1**. [S.l.]: SNIA, 2015.

THOMSON, A.; ABADI, D. J. The case for determinism in database systems. **VLDB Endowment**, v. 3, n. 1-2, p. 70-80, 2010.

TU, S.; ZHENG, W.; KOHLER, E.; LISKOV, B.; MADDEN, S. Speedy transactions in multicore in-memory databases. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 24., 2013, [S.l.]. **Proceedings...** [S.l.: s.n.], 2016. p. 18-32.

WU, Y.; ARULRAJ, J.; LIN, J.; XIAN, R.; PAVLO, A. An empirical evaluation of in-memory multi-version concurrency control. **VLDB Endowment**, v. 10, n. 7, p. 781-792, 2017.

ZHANG, H. *et al*. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 16., 2016, San Francisco. **Proceedings...** San Francisco: [s.n.], 2016. p. 1567-1581. Disponível em: http://doi.acm.org/10.1145/2882903.2915231. Acesso em: 20 ago. 2019.

ZHENG, W.; TU, S.; KOHLER, E.; LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In: USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 11., 2014, Broomfield. **Proceedings...** Broomfield: [s.n.], 2014. p. 465-477. Disponível em: https://www.usenix.org/conference/osdi14/technical-sessions/ presentation/zheng_wenting. Acesso em: 12 ago. 2019.