Capítulo 4

Bancos de dados em memória e suas estratégias de recuperação após falhas

Arlino Magalhães, Ângelo Brayner and José Maria Monteiro

Abstract

In-memory database systems have proven to be an alternative for systems that need massive real-time data processing. In-memory systems keep the database in main memory to provide low latency and high throughput. However, due to memory volatility, these systems are more sensitive to failures than traditional disk databases. Although the components of disk and in-memory databases look similar, these two systems differ greatly in the way they implement their components. This short course provides an overview of the architecture and implementations of in-memory databases and their main recovery strategies after failures. To achieve this goal, this short course provides an overview of in-memory database technology, reviews the concepts of recovery after failures, presents the main architectural choices for implementing in-memory databases and, finally, describes the recovery strategies implemented by a representative sample of modern in-memory databases.

Resumo

Os sistemas de bancos de dados em memória têm se mostrado como uma alternativa para sistemas que precisam de processamento massivo de dados em tempo real. Os sistemas em memória mantêm o banco de dados em memória principal para prover baixa latência e altas taxas de vazão. Contudo, devido à volatilidade da memória, esses sistemas são mais sensíveis a falhas do que os tradicionais bancos de dados em disco. Embora os componentes dos bancos de dados em disco e em memória pareçam similares, esses dois sistemas diferem muito na maneira como implementam os seus componentes. Esse minicurso provê uma visão geral da arquitetura e implementação de bancos de dados em memória e suas principais estratégias de recuperação após falhas. Para atingir essa meta, esse minicurso fornece uma visão geral da tecnologia de bancos de dados em memória, revisa os conceitos de recuperação após falhas, apresenta as principais escolhas

arquiteturais para implementação de bancos de dados em memória e, por fim, descrever as estratégias de recuperação implementadas por uma amostra representativa dos bancos de dados em memória modernos.

4.1. Introdução

Vários cenários de aplicações atuais têm necessitado de um processamento massivo de dados e/ou em tempo real. Os bancos de dados em memória principal têm se mostrado como uma alternativa para tais sistemas. Os sistemas em memória mantêm os dados em memória principal e, por isso, possuem baixa latência e altas taxas de vazão de dados. Essa abordagem fornece muitas implicações importantes em como os componentes desses sistemas devem ser implementados. Por exemplo, os bancos de dados em disco tentam otimizar o acesso ao disco, enquanto os bancos de dados em memória tentam otimizar o acesso à memória [Malviya et al. 2014, Gruenwald et al. 1996, Mohan and Levine 1992].

Bancos de dados em memória são utilizados tipicamente em sistemas estilo OLTP (*Online Transaction Processing* ou Processamento de Transações *Online*). Sistemas OLTP comumente fazem muitas modificações em poucas partes do banco de dados por meio de operações de curta duração. Em contraste, sistemas OLAP (*Online Analytical Processing* ou Processamento Analítico *Online*) tipicamente possuem operações de mais longa duração que manipulam e analisam um grande volume de dados. Eles são usados para dar suporte a *business intelligence*, por exemplo. Comumente, cargas de trabalho OLTP e OLAP são executadas em bancos de dados diferentes: transacional e *data warehouse*, respectivamente. Exisqainda os sistemas HTAP (*Transaction/Analytical Processing* ou Processamento Analítico e de Transações) que precisam de percepções analíticas dos dados mais atuais do banco de dados. Para suportar HTPA, alguns sistemas de bancos de dados em memória permitem cargas de trabalho OLTP e OLAP no mesmo banco de dados [Arulraj et al. 2016a, Copeland and Khoshafian 1985]. A Seção 4.3.1 discute como os sistemas em bancos de dados em memória que suportam HTPA.

A principal vantagem dos sistemas em memória (armazenar os dados na memória principal) também é a principal desvantagem desses sistemas. A volatilidade da memória principal torna os bancos de dados em memória mais sensíveis a falhas que não acontecem nos tradicionais bancos de dados em disco. Por exemplo, todo o banco de dados é perdido em caso de uma falta de energia. Tanto os bancos de dados em disco como os em memória implementam técnicas de *logging*, *checkpoint* e recuperação para prover tolerância a falhas. Contudo, esses dois sistemas diferem muito na maneira como implementam essas técnicas [Faerber et al. 2017, Mohan et al. 1992, Härder and Reuter 1983].

Bancos de dados em memória são pouco abordados em livros e cursos de bancos de dados. Apesar das várias pesquisas e publicações, a recuperação após falhas de bancos de dados ainda é pouco compreendida pela comunidade. Esse minicurso visa elucidar os principais aspectos relacionado a bancos de dados em memória, principalmente os relacionados a recuperação após falhas. Para atingir essa meta, esse minicurso discute o restante dos tópicos como descrito a seguir.

A Seção 4.2 faz uma breve introdução aos sistemas de bancos de dados em memória, um pequeno histórico do desenvolvimento desses sistemas e motivações para o uso deles. Além disso, são discutidas algumas tecnologias emergentes que impulsionaram o

desenvolvimento dos sistemas em memória.

A Seção 4.3 descreve as principais escolhas arquiteturais utilizadas pelos bancos de dados em memória, tais como: armazenamento e organização dos dados, indexação, controle de concorrência e durabilidade e recuperação após falhas. Os tópicos dessa seção são discutidos comparando os bancos de dados em memória com os em disco.

A Seção 4.4 apresenta com detalhes os principais conceitos de recuperação após falhas, focando em bancos de dados em disco. Ela discute o tradicional algoritmo de recuperação ARIES e suas variantes. Além disso, é apresentada a técnica de recuperação instantânea de bancos de dados através de *log* indexado. O objetivo da seção é fornecer os conceitos teóricos básicos necessários para as seções seguintes.

A Seção 4.5 descreve com detalhes as técnicas de *logging*, *checkpoint* e recuperação utilizadas pelos bancos de dados em memória para prover tolerância a falhas. A Seção 4.6 apresenta as principais estratégias implementadas pelas técnicas de tolerância a falhas utilizadas por uma amostra significativa de bancos de dados em memória modernos. Por fim, a seção 4.7 apresenta os principais desafios e direções futuras da pesquisa e desenvolvimento em bancos de dados em memória cujo objetivo é orientar outros pesquisadores.

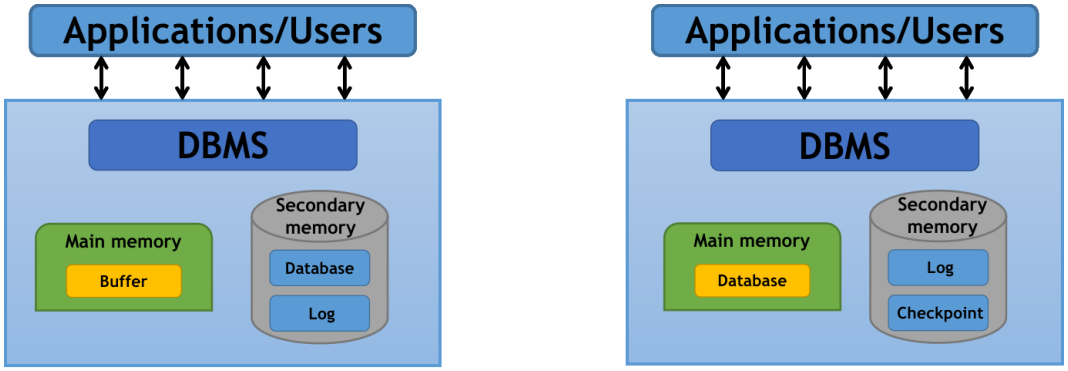
4.2. Bancos de dados em memória

Os Sistemas de Gerenciamento de Bancos Dados - SGBDs (*Database Management Systems* - DBMSs) tradicionais em disco armazenam o banco de dados na memória secundária (*secondary memory*), tais como o disco rígido (*Hard Disk Drive* - HDD) e a unidade em estado sólido (*Solid State Drive* - SSD). Em contraste, os bancos de dados em memória (*Main Memory Databases* - MMDBs ou *In-Memory Databases* - IMDBs) são sistemas cujos dados residem na memória principal (main memory), como a memória RAM (*Random Access Memory* ou Memória de Acesso Aleatório) [Tan et al. 2015, Zhang et al. 2015a, Garcia-Molina and Salem 1992].

A Figura 4.1 (a) ilustra de forma simplificada a arquitetura de um banco de dados em disco. Esse sistema utilizam um mecanismo de *buffering* que copia os dados da memória secundária para a memória principal para que, em seguida, esses dados sejam processados pela CPU (*Central Processing Unit* ou Unidade Central de Processamento). Se necessário, atualizações nos dados devem ser copiadas de volta para a memória secundária. Adicionalmente, esses sistemas mantêm uma cópia do banco de dados em arquivo de *log* para fins de tolerância a falhas [Faerber et al. 2017, Härder and Reuter 1983].

Os bancos de dados em memória (Figura 4.1 (b)) mantêm os dados primariamente e permanentemente na memória principal, eliminando o gargalo de Entrada/Saída - E/S (*Input/Output* - I/O) da memória secundária. Essa abordagem faz esses sistemas muito mais rápidos do que os sistemas em disco. Porém, devido à volatilidade da memória RAM, esses sistemas devem manter uma cópia do banco de dados nos arquivos de *log* e *checkpoint* para prover durabilidade e tolerância a falhas [Faerber et al. 2017, Zhang et al. 2015a, Härder and Reuter 1983].

O cenário dos sistemas de gerenciamento de dados está cada vez mais fragmentado com base em domínios de aplicações. Existem SGBDs em memória relacionais comerciais, como: SAP HANA [Färber et al. 2012], VoltDB [Malviya et al. 2014], Ti-



(a) (b)

Figura 4.1. Arquiteturas dos bancos de dados em disco (a) e em memória (b).

mesTen [Lahiri et al. 2013], SolidDB [Lindström et al. 2013], Hekaton [Larson et al. 2013], Calvin [Thomson et al. 2012] e MemSQL [SingleStore 2022]. Existem também alguns bancos de dados em memória relacionais acadêmicos/*open-source*, tais como: H-Store [Kallman et al. 2008], HyPer [Funke et al. 2014], Silo [Tu et al. 2013], MySQL Cluster NDB [MySQL 2022] e Peloton [Pavlo et al. 2017].

Bancos de dados NoSQL têm se mostrado como uma alternativa mais viável para escalabilidade de bancos de dados, principalmente quando distribuídos em *clusters* [Sadalage and Fowler 2019, Magalhães et al. 2017]. Assim, bancos de dados em memória têm sido desenvolvidos para as categorias NoSQL, como os bancos chave-valor (por exemplo, Redis [Redis 2020], RAMCloud [Ousterhout et al. 2009], MemepiC [Zhang et al. 2015b] e Pilaf [Mitchell et al. 2013]), os orientados a documento (por exemplo, MongoDB [MongoDB Inc. 2022] e Couchbase [Lim et al. 2014]) e os orientados a grafo (por exemplo, Neo4J [Neo4J 2023], Infinite Graph [Infinite Graph 2023] e OrientDB [OrientDB 2023] .

Com o surgimento da era *Big Data*, houve a necessidade de processar quantidades massivas de dados em pequenos intervalos de tempo, com suporte a serviços com latência muito baixa e análise de dados em tempo real. Nesse contexto, foram desenvolvidos bancos de dados em memória para análise de dados que focam no processamento em lote, tais como: Spark [Bishop et al. 2011], SINGA [Ooi et al. 2015], Pregel [Malewicz et al. 2010], GraphLab [Low et al. 2012], Mammoth [Shi et al. 2015], Phoenix [Yoo et al. 2009] e GridGain [GridGain Team 2022]. Além disso, foram desenvolvidos bancos de dados em memória para processamento de dados em tempo real, como: Storm [Apache Software Foundation 2022], Yahoo! S4 [Neumeyer et al. 2010], Spark Streaming [Zaharia et al. 2013] e MapReduce Online [Condie et al. 2010].

4.2.1. Pequeno histórico

A pesquisa e o desenvolvimento de bancos de dados em memória surgiu no início da década de 80, quando foram publicados os primeiros artigos na área, como DeWitt et al. 1984, Hagmann 1986, Eich 1986, Eich 1987a, Lehman and Carey 1987 e Eich 1987b. Em geral, a maioria desses trabalhos focou em melhorar o desempenho dos bancos de dados em disco, assumindo que todo (ou quase todo) o banco de dados cabia na memória

principal. São exemplos de bancos de dados em memória desenvolvidos nessa época: IMS/Fast Path [Strickland et al. 1982], MARS MMDB [Eich 1987b], System M [Salem and Garcia-Molina 1990], TPK [Li and Naughton 1988], OBE [Bitton et al. 1987] e HALO [Garcia-Molina and Salem 1992]. Entretanto, o preço elevado e a capacidade limitada da memória RAM, nessa época, tornou a aplicação de sistemas em memória uma solução inviável para a grande maioria dos problemas.

Nos anos noventa, os avanços na tecnologia geraram novamente interesse na pesquisa em bancos de dados em memória. Dois fatores impulsionaram a volta do interesse em sistemas em memória: preço/capacidade da memória RAM e paralelismo *multicore*. Atualmente, os servidores modernos possuem CPUs que proveem estratégias de processamento em paralelo e podem armazenar um banco de dados inteiro (ou uma parte significativa) em memória a um preço razoável. Além disso, muitos dos servidores contemporâneos possuem vários *sockets*, onde podem ser conectados muitos *terabytes* de DRAM e processadores com centenas de núcleos. Além disso, outras tecnologias recentes têm sido utilizadas para explorar melhor o potencial dos sistemas em memória, tais como: NUMA, SIMD, RDM, HTM e NVRAM [Magalhães et al. 2021b, Zhang et al. 2015a, Garcia-Molina and Salem 1992].

Muitos dos trabalhos desenvolvidos na década de 80 sobre sistemas em memória foram invalidados com as melhorias tecnológicas da década de 90. Os bancos de dados em memória da década de 90 começaram a refletir as tendências e arquiteturas dos sistemas em memória atuais. Alguns produtos comerciais emergiram nesse período sendo empregados em aplicações de desempenho crítico, como telecomunicações, por exemplo. São exemplos desses sistemas de bancos de dados: Dali/DataBlitz [Jagadish et al. 1994], ClustRa [Hvasshovd et al. 1995], TimesTen [Lahiri et al. 2013] e P*Time [Cha and Song 2004].

A volta do interesse em sistemas em memória trouxe uma nova onda de pesquisa e desenvolvimento em bancos de dados em memória. A maioria das empresas desenvolvedoras/vendedoras de bancos de dados atuais possui uma solução de banco de dados em memória, como o TimesTen da Oracle e o Hekaton da Microsoft. Surgiram também *startups*, como VoltDB [VolDB 2022] e MemSQL [SingleStore 2022] (chamado atualmente de SingleStore). O resultado dessas pesquisas e desenvolvimento é um novo tipo de sistema de banco de dados cujo projeto difere radicalmente quando comparado com os tradicionais sistemas em disco. A sessão 4.3 discute a arquitetura e as principais escolhas de implementação de bancos em memória.

4.2.2. Motivação para o uso de bancos de dados em memória

A tecnologia de SGBDs em memória tem se mostrado como uma eficiente alternativa para situações que exigem alto desempenho e/ou visão em tempo-real devido à baixa latência e alta vazão de dados desses sistemas. São exemplos de aplicações contemporâneas com essas exigências: *trading*, publicidade, jogos, previsão do tempo, análise de *big data*, etc. [Magalhães et al. 2018b, Zhang et al. 2015a, Garcia-Molina and Salem 1992].

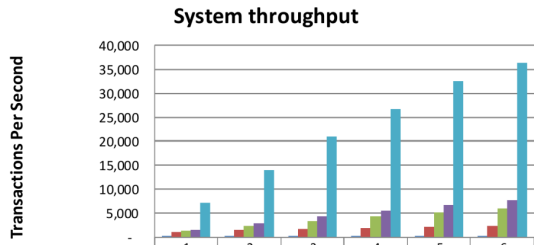
Os sistemas em disco podem não ser capazes de oferecer tempos de resposta aceitáveis para sistemas de desempenho crítico devido à alta latência de acesso à memória secundária. Esse cenário foi inicialmente percebido por empresas de *internet* como Ama-

zon, Google, Facebook e Twitter. Contudo, atualmente outras empresas/organizações têm encontrado esse mesmo obstáculo para fornecer serviços com tempos de resposta aceitáveis. Por exemplo, muitas empresas comerciais precisam detectar uma mudança súbita nos preços de negociação para reagir instantaneamente (em poucos milissegundos) [Magalhães et al. 2018a, Zhang et al. 2015a, Hazenberg and Hemminga 2011].

Melhorias em *hardware* têm provido memórias com alta capacidade de armazenamento a baixo custo. Nas últimas décadas, os *chips* de memória têm dobrado em capacidade de armazenamento a cada 3 anos, enquanto seus preços têm caído em um fator de 10 a cada 5 anos. Como consequência, usar bancos de dados em memória principal para aplicações que tradicionalmente usam bancos de dados em disco tornou-se tecnicamente possível e economicamente viável [Faerber et al. 2017, Zhang et al. 2015a].

O desenvolvimento recente de novas tecnologias têm fornecido ganhos de desempenho com baixa sobrecarga para os sistemas em memória. São exemplos dessas tecnologias: arquitetura NUMA, instruções SIMD, redes RDMA, memória transacional em *hardware* e memória RAM não volátil. Essas tecnologias alavancaram o desenvolvimento de SGBDs em memória [Magalhães et al. 2021b, Faerber et al. 2017, Zhang et al. 2015a]. A seção 4.2.3 detalha o uso dessas tecnologias em sistemas em memória.

Para ilustrar o potencial do uso de bancos de dados em memória, a Figura 4.2 apresenta um experimento de escalabilidade entre o banco em memória Microsoft Hekaton e o tradicional banco em disco Microsoft SQL Server. Em ambos os sistemas, a base de dados estava armazenada inteiramente na memória principal. Os resultados mostram que, por exemplo, com 12 *cores*, Hekaton teve uma vazão de 36.375 transações por segundo, o que correspondeu a um ganho de desempenho quase 16 vezes maior do que o SQL Server, que executou apenas 2.312 transações por segundo. Esse resultado é devido ao fato dos componentes arquiteturais do SQL Server serem projetados para tentar otimizar o acesso à memória secundária, mesmo com a base de dados armazenada inteiramente na memória principal. Por outro lado, os componentes de Hekaton tentam otimizar o acesso à memória principal [Magalhães et al. 2021b, Diaconu et al. 2013]. A seção 4.3 discute os componentes arquiteturais de bancos de dados.



	1	2	3	4	5	6
Number of cores	2	4	6	8	10	12
SQL with contention	984	1,363	1,645	1,876	2,118	2,312
SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
Interop	1,518	2,936	4,273	5,459	6,701	7,709
Native	7,078	13,892	20,919	26,721	32,507	36,375

Figura 4.2. Experimentos de escalabilidade entre o banco de dados em memória Hekaton e o banco de dados em disco SQL Server [Diaconu et al. 2013].

4.2.3. Tecnologias emergentes para bancos de dados em memória

Embora os sistemas em memória tenham surgido nos anos 80, esses sistemas não se consolidaram como uma solução viável devido ao alto preço e limitações de *hardware* da

época. Soluções de *hardware*/arquitetura de sistemas recentes têm sido exploradas para melhoras de ganho de desempenho. Essa seção discute brevemente tecnologias emergentes que alavancaram o desenvolvimento dos bancos de dados em memória [Magalhães et al. 2021b, Magalhaes et al. 2022, Magalhaes et al. 2023, Faerber et al. 2017].

4.2.3.1. Arquitetura NUMA

NUMA (*Non-Uniform Memory Access* ou Acesso não Uniforme a Memória) é uma arquitetura cujo acesso do processador a memória não é uniforme. Nessa arquitetura, o processador pode acessar uma memória local com latência mínima e memórias remotas com latência maior. A Figura 4.3 ilustra uma arquitetura NUMA com 4 nós. Nesse exemplo, no fluxo (1), o processador do *socket* 0 faz um acesso à memória local, que possui latência mínima. No fluxo (2), esse mesmo processador faz um acesso à memória remota no *socket* 3, que possui uma latência maior. Através dessa abordagem, NUMA permite aumentar a largura de banda e o tamanho total de memória que podem ser implementados em um servidor [Li et al. 2013, Zhang et al. 2015a].

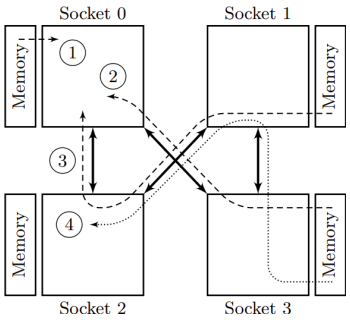


Figura 4.3. Arquitetura NUMA com 4 *sockets* [Li et al. 2013].

Os sistemas de bancos de dados modernos devem ser projetados considerando o uso de NUMA. As pesquisas de bancos de dados em NUMA tentam, por exemplo, minimizar o acesso remoto de dados [Maas et al. 2013, Leis et al. 2014a] e gerenciar os efeitos do acesso NUMA em cargas de trabalho sensíveis à latência (OLTP, por exemplo) [Porobic et al. 2012, Porobic et al. 2014].

4.2.3.2. Instruções SIMD

SIMD (*Single Instruction Multiple Data* ou Única Instrução Múltiplos Dados) é um conjunto de instruções que está presente nos processadores atuais e provê uma fácil alternativa para execução paralela ao nível de dados. Nessa abordagem, uma mesma instrução pode aplicar várias operações simultaneamente a diversos dados para produzir vários resultados. A Figura 4.4 (a) mostra a execução de uma instrução com uma única operação sobre os dados X_0 e Y_0 para produzir o resultado Z_0. Em contraste, a Figura 4.4 (b) mostra a execução de uma instrução SIMD sobre *n* dados X e *n* dados Y para produzir *n* resultados Z. As *n* operações são executadas em paralelo no processador. Como desvantagens, uma

instrução SIMD possui um limite de paralelismo permitido e restrições na estrutura de dados em que pode operar [Willhalm et al. 2009, Neumann 2011].

Um projeto eficiente de bancos de dados deve considerar paralelismo ao nível de dados. Instruções SIMD podem acelerar operações caras de bancos de dados, como junções e ordenações [Chhugani et al. 2008,Neumann 2011]. O banco de dados SAP HANA, por exemplo, possui um esquema de vetor que utiliza instruções SIMD para acelerar operações de *scan* (varredura) [Willhalm et al. 2013].

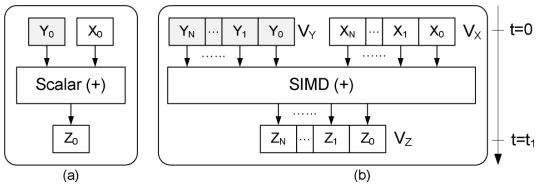


Figura 4.4. Modo escalar (a): uma operação produz um resultado. Modo SIMD (b): uma operação produz vários resultados [Willhalm et al. 2009].

4.2.3.3. Redes RDMA

RDMA (*Remote Direct Memory Access* ou Acesso Remoto Direto à Memória) permite a transmissão de dados entre as memórias de dois computadores sem envolver os sistemas operacionais das duas máquinas. Diferentemente da tradicional rede *ethernet*, um cliente pode acessar a memória de um servidor diretamente, sem a coordenação desse último. Essa transmissão de dados não requer o uso de CPU, *cache* e troca de contexto. Como o servidor utiliza menos recursos, ele pode direcioná-los para outros fins [Mitchell et al. 2013].

RDMA permite um grande tráfego de dados com baixa latência. Porém, essa estratégia é ineficiente na coordenação e sincronização de múltiplos acessos à memória remota [Mitchell et al. 2013]. O bando de dados FaRM implementa leituras livres de bloqueio através de RDMA [Dragojevic et al. 2014]. HyPer faz *backups* do banco de dados via RDMA para livrar o servidor do trabalho de transmissão de dados [Kemper and Neumann 2011].

4.2.3.4. Memória transacional em *hardware*

HTM (*Hardware Transactional Memory* ou Memória Transacional em *Hardware*) provê um mecanismo de controle de concorrência, semelhante ao mecanismo utilizado pelas transações de banco de dados, que visa aumentar o paralelismo e minimizar conflitos. A Figura 4.5 esboça os benefícios de HTM em relação a outros mecanismos de controle de concorrência implementados em SGBDs em memória. Nesse exemplo, HTM (em vermelho) teve uma vazão de transações superior à execução serial (em amarelo) e ao bloqueio de duas fases (em verde). HTM apresenta um desempenho próximo ao do esquema de execução serial em particionamento estático otimizado (em azul), que frequentemente é muito difícil de implementar [Herlihy and Moss 1993, Karnagel et al. 2014, Leis et al. 2014b, Makreshanski et al. 2015]. A Seção 4.3.3 discute os protocolos de controle de concorrência com mais detalhes.

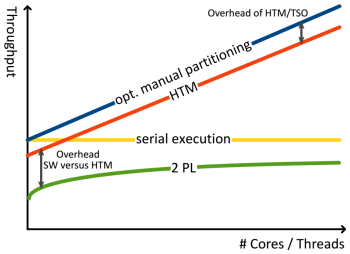


Figura 4.5. HTM *versus* bloqueio em duas fases, execução serial e particionamento estático [Leis et al. 2014b].

A ideia básica de HTM é usar a *cache* como *buffer* local de transação para prover isolamento e detectar conflitos. Para ilustrar essa abordagem, suponha que a Transação 1 (Figura 4.6 (a)) e a Transação 2 (Figura 4.6 (b)) são executadas simultaneamente. Essas duas transações possuem operações em conflito, uma vez que elas atualizam o mesmo objeto a. Durante a execução, as duas transações registram os valores a serem armazenados na memória principal em *cache* temporariamente. Os conflitos são verificados e tratados na hora do *commit* (finalização) da transação. Caso ocorra algum conflito, apenas uma das transações é finalizada. A outra transação deve ser forçada a abortar e reiniciar [Leis et al. 2014b].

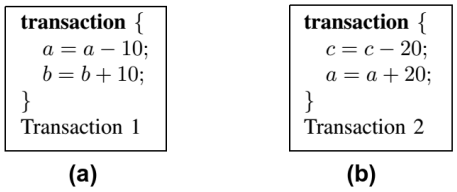


Figura 4.6. Duas transações com operações em conflito [Leis et al. 2014b].

HTM é semelhante ao mecanismo de controle de concorrência implementado pelos SGBDs. Contudo, HTM permite que transações executem com menos sobrecarga, uma vez que a concorrência é gerenciada pelo *hardware*. HTM possui algumas desvantagens: as transações devem ter o tamanho limitado ao da *cache* e transações podem ser abortadas devido a falsos conflitos, troca de contexto, interrupções ou *page faults* [Hammond et al. 2004, Harris et al. 2007]. Os bancos de dados HyPer e DBX, por exemplo, suportam HTM para controle de concorrência [Leis et al. 2014b, Wang et al. 2014].

4.2.3.5. Memória RAM não volátil

NVRAM (*Non-Volatile Random Access Memory* ou Memória RAM não volátil) é uma tecnologia que possui a alta velocidade e endereçamento ao nível de *bytes* da RAM com a persistência e grande capacidade de armazenamento do disco. Contudo, as arquiteturas existentes para bancos de dados em disco e em memória são inapropriados para um banco de dados em NVRAM. Essas duas primeiras utilizam uma estratégia de propagação de dados entre as memórias principal e secundária para fins de durabilidade. Além disso, os dados são mantidos em duas cópias (banco de dados e *log*) com o objetivo de prover tolerância a falhas. Essas estratégias de propagação e duplicação de dados são desnecessárias

em um banco projetado em NVRAM e, ainda, podem causar degradação no desempenho do sistema. As características de NVRAM podem remover os custos causados por essas duas estratégias [Arulraj et al. 2015, Arulraj and Pavlo 2017].

Embora existam muitas pesquisas que descrevem a arquitetura de um SGBD em NVRAM (por exemplo, van Renen et al. 2018, Oukid et al. 2017, Arulraj and Pavlo 2017, Harris 2016, Arulraj et al. 2016b, Garg et al. 2015, Arulraj et al. 2015, [Schwalb et al. 2015], Zhang et al. 2015c, Chen et al. 2011a), ainda não está claro qual o melhor projeto para um SGBD que utiliza NVRAM. Bancos de dados em NVRAM não são o foco desse minicurso e, por isso, não serão mais discutidos.

4.3. Projeto de bancos de dados em memória

Uma suposição simples de como implementar um banco de dados em memória seria ter um SGBD em disco com memória principal suficiente para armazenar a base de dados inteira. Essa estratégia melhora o desempenho do sistema, pois diminui o acesso à memória secundária. Porém, apenas trocar a camada de armazenamento do disco para a memória não torna um SGBD em disco tão eficiente quanto um SGBD em memória. Os SGBDs em disco focam em otimizar o acesso à memória secundária. Por outro lado, os SGBDs em memória são projetados para tentar otimizar o acesso à memória principal. Essa abordagem fornece muitas implicações importantes em como os componentes arquiteturais dos SGBDs em memória devem ser implementados [Magalhães et al. 2021b, Magalhaes et al. 2022, Magalhaes et al. 2022]. As abordagens de projeto utilizadas pelos SGBDs em memória são diversas. Essa seção discute as principais escolhas de implementação e componentes arquiteturais dos SGBDs em memória.

4.3.1. Armazenamento de dados

Os bancos de dados em disco utilizam um mecanismo de *buffer* para acessar registros na memória secundária. Quando é necessário acessar um registro no banco de dados, o gerenciador de *buffer* verifica se a página do banco de dados que contém esse registro está na memória. Caso contrário, ele deve copiar o bloco (ou blocos) do disco, onde a página está armazenada, para a memória principal. A página é a unidade de leitura/gravação de informação no banco de dados e pode conter várias linhas de dados. Uma página possui uma representação semelhante ao bloco do disco para evitar traduções entre as duas representações. Com a página na memória principal, o gerenciador de *buffer* ainda deve fazer um acesso indireto ao registro através dos seguintes passos: (1) encontrar a página no *buffer* e (2) calcular o deslocamento necessário para acessar o registro na página [Härder and Reuter 1983, Ramakrishnan and Gehrke 2003].

A Figura 4.7 esquematiza o funcionamento do gerenciador de *buffer*. O *buffer* funciona como um quadro de páginas na memória principal. Ele ainda implementa algum mecanismo para acessar as páginas mais rapidamente na memória principal, como uma tabela *hash*. O gerenciador de *buffer* é uma solução simples e elegante. Através desse mecanismo, os outros componentes do SGBD abstraem como os dados são acessados no disco. Contudo, o acesso indireto de registros no *buffer* pode afetar o desempenho dos sistemas em memória. Os SGBDs em memória não são dependentes do formato dos blocos de disco. Assim, as suas representações de dados são implementadas de forma a

levar ao melhor desempenho no sistema [Magalhães et al. 2021b, Faerber et al. 2017, Hazenberg and Hemminga 2011].

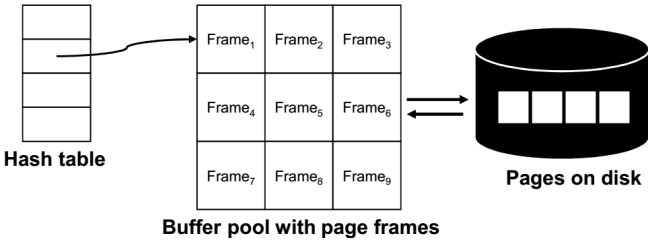


Figura 4.7. Gerenciador de *buffer* em um SGBD em disco [Faerber et al. 2017].

Os bancos de dados em memória comumente implementam ponteiros para acesso direto a memória. Como os ponteiros evitam a sobrecarga do acesso indireto a registros, o sistema utiliza menos ciclos de CPU [Larson and Levandoski 2016,Faerber et al. 2017]. O uso de ponteiros pode melhorar o acesso à memória em ordens de magnitude. Experimentos realizados no IBM Starburst evidenciam essa melhoria. Os experimentos compararam os componentes de banco de dados em memória e em disco desse sistema. Os resultados mostraram que o gerenciador de *buffer* foi responsável por até 40% do tempo de execução de uma consulta, mesmo com as bases de dados dos dois componentes armazenadas inteiramente na memória principal [Lehman et al. 1992].

4.3.1.1. Particionamento de dados

Alguns SGBDs, como H-Store, VoltDB e Calvin, usam um esquema de particionamento de dados. Uma partição divide um banco de dados em partes disjuntas e independentes. As transações são executadas serialmente em partições de dados. Essa estratégia tem a vantagem de uma transação poder executar com acesso exclusivo aos recursos do sistema sem a sobrecarga do controle de concorrência, uma vez que a transação executa sozinha nas partições que precisa. Transações diferentes ainda podem executar em paralelo em partições diferentes usando núcleos de CPU diferentes. Além disso, partições podem ser armazenadas em máquinas diferentes, permitindo ao sistema possuir mais memória do que seria disponível em apenas uma máquina [Faerber et al. 2017, Taft et al. 2014, Thomson et al. 2012].

A desvantagem do esquema de particionamento ocorre quando uma transação precisa acessar mais de uma partição e alguma delas não está disponível. Assim, a transação deve esperar até que todas as partições que precisa estejam disponíveis. Além disso, essa estratégia precisa de balanceamento de carga em partições muito frequentemente acessadas. Em contraste, as transações em sistemas não participados (por exemplo, Hekaton, SAP HANA, MemSQL e TimesTen) podem acessar qualquer parte do banco de dados [Funke et al. 2014, Malviya et al. 2014, Thomson et al. 2012].

A Figura 4.8 ilustra um esquema de tabelas particionadas. As tabelas *A*, *B* e *C* estão divididas nas partições *X*, *Y* e *Z*. Por exemplo, a tabela *A* está dividida em partes *A'*, *A"* e *A"'* armazenadas nas partições *X*, *Y* e *Z*, respectivamente. Consultas podem executar em paralelo em partições diferentes. Por exemplo, enquanto uma consulta está percorrendo *A'* em *X*, outra consulta pode percorrer *A"* em *Y* ao mesmo tempo. Porém,

uma consulta que precise percorrer a tabela *A* inteira deve esperar até que as três partições estejam disponíveis.

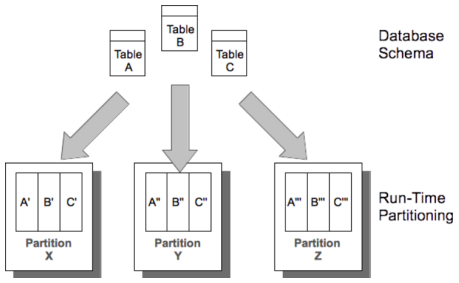


Figura 4.8. Esquema de particionamento de tabelas de bancos de dados [VoltDB Documentation 2022].

4.3.1.2. Versionamento de dados

Muitos SGBDs em memória implementam o multi versionamento de dados, como Hekaton, HyPer e SAP HANA. Nessa abordagem, uma modificação em um dado não sobrescreve o seu conteúdo. A modificação é escrita para outra versão do dado, chamada de versão sombra (*shadow version*), em um endereço de memória diferente do original. Quando a modificação é finalizada, a versão sombra se torna a versão atual do banco de dados. A versão anterior permanece no banco de dados, sendo removida apenas posteriormente por algum sistema de coleta de lixo (*garbage collection*) [Chen et al. 2011b, Malviya et al. 2014, Neumann et al. 2015].

O versionamento de dados facilita a implementação de protocolos de controle de concorrência sem bloqueios. Os protocolos de bloqueio podem ser uma fonte de sobrecarga para sistemas em memória. A Seção 4.3.3 explica os mecanismos de controle de concorrência em mais detalhes. O versionamento permite ainda uma maneira fácil de criar *backups* do banco de dados para fins de tolerância a falhas [Chen et al. 2011b, Malviya et al. 2014, Neumann et al. 2015]. A Seção 4.5 explica melhor como os sistemas em memória proveem tolerância a falhas. O versionamento de dados também facilita a criação de sistemas HTAP [Arulraj et al. 2016a, Copeland and Khoshafian 1985].

A Figura 4.9 esboça um esquema de armazenamento multi versionado. Esse exemplo representa uma tabela simples de contas bancárias. As versões dos dados podem ser acessadas por meio de uma tabela *hash*. Por exemplo, o *bucket J* da *hash* aponta para uma lista que contém quatro versões, mas apenas duas delas são válidas. As colunas *Begin* (início) e *End* (fim) da tabela indicam o intervalo de tempo de validade de cada versão de um registro. Assim, a versão (Jonh, Londom, 100) começou a ser válida no tempo 10 e se tornou obsoleta no tempo 20. O registro (Jonh, Londom, 130) é uma versão válida atual, pois seu tempo final de validade possui o valor *inf*, de *infinite* (infinito) [Diaconu et al. 2013].

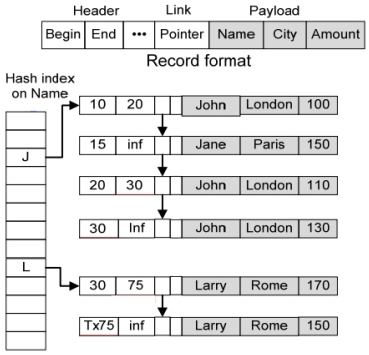


Figura 4.9. Esquema de versionamento de dados [Diaconu et al. 2013].

4.3.1.3. Organização de dados

Os SGBDs em memória podem ser orientados a linhas (*N-ary Storage Model* - NSM ou Modelo de Armazenamento N-ário). Nesse modelo, os atributos dos registros do banco de dados são armazenados contiguamente. Esse *layout* é mais adequado para sistemas OLPT cujas transações tendem a manusear os atributos de determinados registros por vez. A Figura 4.10 (a) ilustra uma tabela orientado a linhas. Por exemplo, todos os atributos do registro de *ID* 101 estão armazenados um após o outro, seguidos pelos atributos do registro de *ID* 102 [Arulraj et al. 2016a, Copeland and Khoshafian 1985].

Os SGBDs em memória podem ser orientados a colunas (*Decomposition Storage Model* - DMS ou Modelo de Armazenamento de Decomposição). O modelo de organização colunar armazena os valores de um atributo de todos os registros contiguamente. Esse modelo é comumente empregado em sistemas OLAP. Geralmente, uma transação nesse tipo de sistema acessa um atributo (ou um subconjunto de atributos) de muitos registros ao mesmo tempo. A Figura 4.10 (b) exemplifica uma tabela orientado a colunas. Por exemplo, todos os valores da coluna *ID* são armazenados um após o outro, seguidos pelos valores da coluna *IMAGE-ID* [Copeland and Khoshafian 1985, Arulraj et al. 2016a].

ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

(a)

ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

(b)

Figura 4.10. Modelos de tabelas orientadas a linhas (a) e a colunas (b) [Arulraj et al. 2016a]

Os SGBDs em memória são comumente empregados em sistemas OLTP pelo fato de possibilitarem um acesso rápido a registros individuais. Porém, alguns sistemas possuem características HTAP, ou seja, precisam de percepções analíticas dos dados mais atuais do banco de dados. Para esses sistemas, o modelo FSM (*Flexible Storage Model* ou Modelo de Armazenamento Flexível) é o mais adequado. O FSM é uma generalização dos modelos NSM e DSM, ou seja, ele suporta tanto o modelo orientado a linhas como o colunar no mesmo banco de dados. Geralmente, FSM armazena os dados primeiramente

no formato orientado a linhas. Quando um dado se torna pouco (ou nunca) acessado, ele é reorganizado para o formato colunar [Copeland and Khoshafian 1985, Arulraj et al. 2016a].

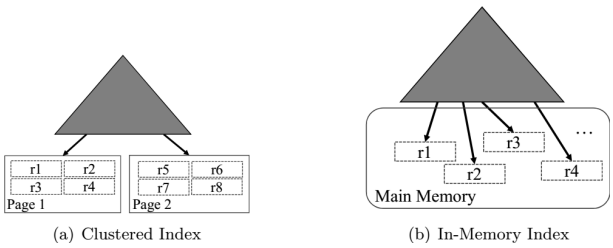
HyPer usa um esquema de memória virtual para separar os dados entre os mais imutáveis (para transações OLAP) e os mais recentemente atualizados (para transações OLTP), ver Seção 4.6.1.3 para mais detalhes [Funke et al. 2014]. SAP HANA implementa uma estrutura de tabela unificada para suportar HTAP, ver Seção 4.6.1.4 para mais detalhes [Sikka et al. 2012].

4.3.2. Indexação

Um índice é uma estrutura de dados que organiza os registros do banco de dados para otimizar determinados tipos de operações de busca. Eles permitem rápido acesso aos dados sem a necessidade de percorrer uma tabela inteira. Os índices nos bancos de dados em disco são mapeados para páginas gerenciadas pelo *buffer* visando minimizar o acesso à memória secundária. A maioria dos bancos de dados em disco suportam índices em árvores B^+. Sistemas em disco também utilizam índices clusterizados, que mantêm os registros em uma ordem específica no disco. Esses dois tipos de índice são eficientes em buscas usando faixas de valores. Uma alternativa é o índice com tabela *hash*, muito eficiente em buscas com um valor específico [Ramakrishnan and Gehrke 2003, Elmasri and Navathe 2000].

Embora os sistemas em memória tenham altas taxas de vazão de dados, eles ainda precisam de índices para acelerar o acesso a dados. Algumas técnicas de indexação tentam otimizar o acesso a *cache*, como: HOT [Binna et al. 2018], CSS-Trees [Rao and Ross 2000], CSB-Trees [Rao and Ross 2000], pB-Trees [Chen et al. 2001], FAST [Kim et al. 2010] e ART [Leis et al. 2013]. Algumas estratégias de indexação consideram o paralelismo *multi-core*, como: MRBTree [Pandis et al. 2011], Bw-tree [Levandoski et al. 2013] e Mass-Tree [Mao et al. 2012]. Os sistemas em memória também usam índices de tabela *hash* [Fan et al. 2013].

Como os bancos de dados em memória não implementam um *buffer*, os seus índices armazenam ponteiros direto para registros, ao invés de *IDs* ou chaves primárias (como nos bancos de dados em disco). A Figura 4.11 (a) representa uma indexação de SGBDs em disco cujos índices apontam para páginas. Em contraste, a Figure 4.11 (b) representa uma estrutura de índice de SGBDs em memória que possui ponteiros direto para registros [Faerber et al. 2017, Ailamaki et al. 1999].



(a) Clustered Index (b) In-Memory Index

Figura 4.11. Estruturas de índices nos bancos de dados em disco (a) e em memória (b) [Faerber et al. 2017].

4.3.3. Controle de concorrência

Controle de concorrência é um método usado para impedir que transações acessem o mesmo dado simultaneamente e levem o banco de dados a uma inconsistência. Comumente, os bancos de dados em disco utilizam algum controle de concorrência baseado em bloqueio, como o bloqueio em duas fazes (*Two-Phase Locking* - 2PL) para garantir o controle de concorrência. Nesse protocolo, por exemplo, uma transação T_1 deve adquirir bloqueios nos dados que necessita antes de começar a modificá-los. Uma transação T_2 que necessite alterar dados bloqueados por T_1 deve esperar até que T_1 não precise mais desses dados e desbloqueie-os [Weikum and Vossen 2002, Ramakrishnan and Gehrke 2003].

O gerenciador de bloqueio realiza as operações de bloqueio e desbloqueio atomicamente. Além disso, ele ainda deve verificar se um dado está bloqueado sempre que uma transação solicita acesso a esse dado. Essas operações são toleradas em bancos de dados em disco, visto que o acesso ao disco é o maior gargalo. Porém, bloqueios são muito custosos em sistemas em memória, pois acrescentam ciclos de CPU. Por esse motivo, os sistemas em memória evitam implementar um gerenciador de bloqueio [Weikum and Vossen 2002, Hazenberg and Hemminga 2011].

Alguns SGBDs em memória usam metadados embutidos em registros para controlar o acesso a eles. Por exemplo, um *bit* de bloqueio é embutido no registro para informar se ele está bloqueado ou não [Garcia-Molina and Salem 1992]. Uma alternativa é usar um número contador para representar a quantidade de requisições de bloqueio feitas por transações a um registro [Ren et al. 2012].

Muitos SGBDs em memória adotam um modelo de controle de concorrência multi-versionado (*Multi-Version Concurrency Control* - MVCC). Nesse modelo, transações de apenas leitura não precisam bloquear dados, visto que transações de escrita fazem suas modificações em versões diferentes dos dados lidos. Como desvantagem, MVCC possui a sobrecarga de criar novas versões de dados e de remover versões obsoletas [Neumann et al. 2015, Lomet et al. 2012].

Hekaton and HyPer implementam um MVCC otimista. Essa abordagem não utiliza bloqueios e conflito são verificados apenas na finalização da transação. Se algum conflito for detectado, apenas uma das transações envolvidas pode finalizar, as demais devem ser abortadas [Neumann et al. 2015, Lomet et al. 2012]. SAP HANA implementa um MVCC pessimista. Nesse método, MVCC é utilizado apenas em operações de leitura. Em caso de operações de escrita, bloqueios ao nível de registro são usados [Lee et al. 2013].

Sistema que usam particionamento de dados (H-Store, VoltDB e Calvin, por exemplo) não precisam implementar um mecanismo de controle de concorrência, pois as transações executam serialmente, como descrito na Seção 4.3.1. Porém, como desvantagem, transações devem esperar até que todas as partições que precisam estejam disponíveis [Kallman et al. 2008, Thomson et al. 2012, Stonebraker and Weisberg 2013].

4.3.4. Processamento de consultas

O processamento de consultas visa extrair os dados do banco de dados da maneira mais eficiente possível. Para isso, o processador de consultas realiza as etapas de análise, otimi-

zação e execução de consulta. Os SGBDs em disco e memória realizam as atividades de análise e otimização de forma semelhante. Porém, eles diferem muito na implementação de execução de consultas [Silberschatz et al. 2020, Ramakrishnan and Gehrke 2003, Hazenberg and Hemminga 2011].

O processamento de consultas nos bancos de dados em disco foca em minimizar as E/Ss no disco. Comumente, esses sistemas utilizam algum modelo de interação estilo Volcano [Graefe and McKenna 1993]. Esse modelo é bem simples e utiliza uma combinação variada de operadores implementados de forma genérica para executar uma consulta [Hazenberg and Hemminga 2011, Zhang et al. 2015a, Faerber et al. 2017].

O modelo de interação é ineficiente em SGBDs em memória. Devido à natureza genérica, um operador deve ser interpretado em tempo de execução e ainda pode ser chamado várias vezes. Por esse motivo, o modelo de interação leva a sobrecargas desnecessárias nos bancos de dados em memória. Assim, esses sistemas preferem compilar transações em código de máquina para evitar a interpretação em tempo de execução e, consequentemente, fazer melhor uso de memória e CPU. VoltDB implementa transações compiladas em código de máquina através de procedimentos armazenados (*stored procedures*). Como desvantagem dessa abordagem, o sistema precisa conhecer as transações com antecedência, o que pode ser difícil de implementar em alguns sistemas. Por exemplo, as consultas de sistemas *ad hoc* são criadas a partir de um requisito específico do momento em que são criadas [Stonebraker and Weisberg 2013, Diaconu et al. 2013, Kemper and Neumann 2011, Menon et al. 2017].

4.3.5. Durabilidade e recuperação após falhas

A maioria dos bancos de dados em disco utilizam alguma forma de recuperação após falhas baseada no protocolo ARIES. ARIES escreve informações de atualizações em páginas para um arquivo de *log* na memória secundária. O *log* possui informação suficiente para desfazer ou refazer uma atualização inconsistente. Além disso, *checkpoints* são realizados para identificar atualizações de transações que já foram persistidas em disco e, consequentemente, diminuir a quantidade de informação no *log* a ser processada em uma recuperação após falha [Mohan and Levine 1992]. A Seção 4.4 detalha a recuperação de SGBDs em disco.

Os bancos de dados em memória também realizam atividades de *logging* e *checkpoint* para fins de tolerância a falhas. Além disso, essas técnicas também proveem durabilidade, uma vez que a memória principal é volátil. Embora os componentes desses dois sistemas pareçam semelhantes, eles diferem muito na maneira como são implementados [Magalhães et al. 2021b, Magalhaes et al. 2022, Magalhaes et al. 2022]. A Seção 4.5 discute a recuperação de SGBDs em memória com mais detalhes.

4.4. Recuperação em bancos de dados em disco

Os gerenciadores de *buffer*, comumente, o utilizam os protocolos *No-force* e *Steal* em suas políticas de substituição de páginas. No protocolo *No-force*, uma página "suja" não precisa ser enviada para a memória secundária imediatamente à finalização de sua transação. Uma página é considerada "suja"quando ela possui modificações na memória principal que ainda não foram copiadas para a memória secundária. *No-force* permite que pági-

nas muito atualizadas permaneçam no *buffer*, evitando múltiplas E/Ss para a memória secundária, as quais são operações caras. Em *Steal*, páginas "sujas" de transações ativas (transações em execução) podem ser movidas para a memória secundária antes da finalização da transação. Com *Steal*, o sistema pode ter um banco de dados maior do que a memória disponível, pois páginas de transações ativas podem ser movidas para o disco visando dar espaço na memória para novas páginas [Härder and Reuter 1983, Mohan et al. 1992].

As falhas (*crashes*) de bancos de dados são eventos que afetam o processamento do sistema, tais como: *bugs* de *software*, entradas de dados erradas, defeitos de *hardware*, falta de energia, erros humanos, dentre outros. Após um *crash*, o gerenciador de recuperação deve assegurar que o banco de dados não fique em um estado inconsistente. Uma transação pode ser considerada a unidade de recuperação em um banco de dados. Assim, o gerenciador de recuperação deve assegurar que as transações finalizadas tenham suas páginas escritas na memória secundária, mesmo que essas páginas não tenham sido movidas para o disco antes da falha. Além disso, deve ser assegurado que transações não finalizadas não tenham suas modificações refletidas na memória secundária. Esses dois cenários de inconsistência são possíveis devido às políticas de substituição de páginas utilizadas pelo gerenciador de *buffer* [Härder and Reuter 1983, Mohan et al. 1992].

Basicamente, existem as falhas de transação, sistema e disco. A falha de transação ocorre quando uma transação não pode atingir suas metas ou viola alguma integridade do banco de dados (duplicação de chave primária, por exemplo) então deve ser cancelada e desfeita. Esse tipo de falha não interrompe a operação do sistema e nem atrapalha o processamento das outras transações. A falha de sistema acontece quando o conteúdo da memória principal é perdido (falta de energia, por exemplo), impossibilitando que o sistema continue operando. A falha de disco acontece quando a memória secundária é perdida (defeito de disco, por exemplo), impedindo o funcionamento do sistema. Nessa falha, diferentemente das outras duas, é necessário reparar ou trocar o dispositivo de armazenamento defeituoso antes de começar o processo de recuperação [Mohan et al. 1992].

A Figura 4.12 exemplifica uma falha de sistema. Nesse cenário, cinco transações executam até acontecer a falha de sistema. Após esse tipo de falha, o sistema não pode continuar operando e deve ser reiniciado. Assim que o sistema reinicia, o gerenciador de *buffer* começa o processo de recuperação do banco de dados. As modificações das transações T1, T2 e T3 devem ser persistidas, uma vez que elas finalizaram. Assim, essas transações precisam ser refeitas, se necessário. Como as transações T4 e T5 não finalizaram, as suas modificações devem ser desfeitas [Härder and Reuter 1983].

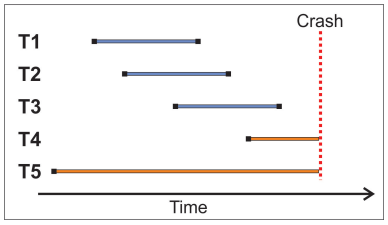


Figura 4.12. Um cenário de falha de sistema [Härder and Reuter 1983].

4.4.1. O algoritmo de recuperação ARIES

ARIES (*Algorithms for Recovery and Isolation Exploiting Semantics*) é um método de recuperação após falhas bastante conhecido e utilizado pelos bancos de dados. A maioria dos SGBDs em disco usa algum método de recuperação estilo ARIES. Esse método de recuperação suporta *Steal*, *No-force*, WAL, *checkpoint fuzzy*, registro de compensação de *log* e granularidade fina [Mohan et al. 1992].

4.4.1.1. *Write-ahead logging*

O protocolo WAL (*Write-Ahead Logging*) assegura que, antes de uma modificação no bando de dados, seja registrada uma informação que descreve essa modificação em um arquivo de *log* na memória secundária. O *log* é um arquivo cujos registros são armazenados sequencialmente. Cada registro possui um LSN (*Log Sequence Number*) que é um número em ordem crescente que funciona como uma identificação única. Geralmente, os bancos de dados em disco utilizam *log* físico cujos registros armazenam informações físicas dos dados atualizados (páginas, por exemplo). Um registro de *log* armazena a imagem de antes (*before image*) e a imagem de depois (*after image*) da modificação de um dado [Mohan et al. 1992].

SGBDs comerciais, como MySQL [MySQL 2020] e Oracle [Oracle 2020], tipicamente implementam um *log* fisiológico por questões de desempenho. No *log* fisiológico, um registro referencia uma página e uma operação lógica na página. Por exemplo, um registro pode conter a imagem da página de antes da atualização e a operação de atualização na página [Mohan et al. 1992, Tucker 2004, Gray and Reuter 1993].

Os registros de *log* mapeiam todas as atualizações feitas no banco de dados. Assim, após um *crash*, os registros de *log* são utilizados para refazer (REDO) ou desfazer (UNDO) modificações de transações que tenham gerado alguma inconsistência no banco de dados. Os registros de *log* de uma determinada transação são ligados, como em uma lista encadeada, permitindo percorrê-los para refazer ou desfazer a transação individualmente. O arquivo de *log* deve ser copiado para discos diferentes para sobreviver a falhas de disco e, se possível, armazenado em locais diferentes para sobrevier a catástrofes [Mohan et al. 1992].

4.4.1.2. *Checkpoint fuzzy*

Como o *log* é um arquivo em crescimento, ele deve ser truncado periodicamente. Essa ação é necessária, pois o disco possui espaço limitado. Alguns sistemas tendem a ter um arquivo de *log* muito maior do que o banco de dados, como os sistemas OLTP que fazem muitas modificações em poucas partes do banco de dados. Além disso, quanto mais registros existirem no *log*, mais informação deverá ser processada durante uma recuperação e, consequentemente, a recuperação será mais lenta. Assim, *checkpoints* devem ser executados periodicamente para encontrar um novo ponto no *log* de onde a recuperação pode começar. Esse ponto é um registro no *log* onde se tem a certeza de que todas as modificações de transações finalizadas anteriormente ao ponto foram persistidas na memória secundária. Assim, os registros anteriores ao *checkpoint* podem ser descartados do

log [Mohan et al. 1992].

ARIES usa o *checkpoint fuzzy*. Essa técnica copia informações de ocupação do *buffer* durante o processamento de transações, como páginas "sujas" e transações ativas. Essas informações são utilizadas para encontrar o registro no *log* de onde a recuperação deve iniciar. *Fuzzy* possui a vantagem de não interferir no processamento das transações. Contudo, ele torna o processo de recuperação mais lento, pois é necessário analisar o *log* para encontrar o registro de início da recuperação [Mohan et al. 1992].

A Figura 4.13 ilustra um cenário de execução do *checkpoint fuzzy*. As páginas P1, P2, P3 e P4 possuem escritas de registros de atualização no arquivo de *log*, representadas pelas linhas pontilhadas. Por exemplo, a página P1 armazenou registros de *log* com LSNs 30, 60 e 100. Durante a execução do *checkpoint*, informações sobre as páginas "sujas" P1, P3 e P4 são armazenadas no arquivo de *checkpoint*. Informações sobre a página P2 não são armazenadas, uma vez que essa página foi salva na memória secundária antes do início do *checkpoint*. Adicionalmente, o *checkpoint* também armazena informações sobre as transações ativas, que não estão representadas nesse exemplo. Após uma falha, as páginas P1, P3 e P4 são identificadas como "sujas". Com essa informação o sistema pode identificar que a recuperação deve iniciar a partir do registro de LSN 20, pois ele é o mais antigo entre os registros das páginas "sujas". Além disso, as transações que estavam ativas também são identificadas [Magalhães 2022, Mohan et al. 1992, Härder and Reuter 1983].

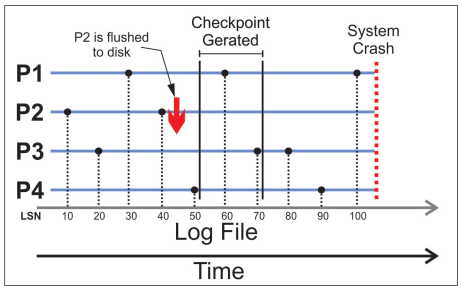


Figura 4.13. Um cenário de execução de *checkpoint fuzzy* [Magalhães 2022].

4.4.1.3. Recuperação após falha de sistema

Após uma falha de sistema, ARIES recupera um banco de dados por meio de três fases: Análise, Refazer e Desfazer [Mohan et al. 1992].

Assim que o sistema reinicia após uma falha de sistema, o gerenciador de recuperação examina o último *checkpoint* para obter as informações gravadas de transações ativas e páginas "sujas". Em seguida, a fase de análise percorre o arquivo de *log* começando do último *checkpoint* até o final do arquivo para encontrar todas as transações ativas e páginas "sujas" até o momento do *crash*. Com essas informações, o sistema calcula de onde a fase Refazer deve começar [Mohan et al. 1992].

A fase Refazer percorre o arquivo de *log* começando do primeiro registro de atualização das páginas "sujas" até o final do arquivo. O objetivo dessa fazer é refazer todas as transações ativas com páginas "sujas"de antes da falha. Esse método é chamado de pa-

radigma de repetição da história, pois reconstrói o banco de dados para o mesmo estado em que estava no momento da falha [Mohan et al. 1992, Mohan 1999].

A fase Desfazer percorre o arquivo de *log* em ordem inversa começando do último registro até desfazer todas as transações que não finalizaram até o *crash*. Cada transação é desfeita em ordem inversa de execução de seus registros de *log*. Assim que todas as transações não finalizadas forem desfeitas, o processo de recuperação termina e o banco de dados pode processar novas transações [Mohan et al. 1992].

4.4.1.4. Registro de *log* de compensação

ARIES introduz o conceito de registro de *log* de compensação (*Compensation Log Record* - CLR). Para cada ação de registro de *log* desfeita, um CLR é armazenado no arquivo de *log*. Um CLR descreve as ações executadas durante uma operação de desfazer (*rollback*). Esse tipo registro nunca é desfeito, ou seja, nunca acontece um desfazer de um registro de *rollback*. O CLR possui um campo que aponta para o registro de sua transação que é anterior ao registro desfeito. Assim, durante o desfazer de uma transação, não é necessário desfazer o CLR e nem o registro cujo *rollback* é descrito por esse CLR. Isso é possível porque esses dois registros possuem ações opostas e desfazê-las um após a outra não traz efeito algum ao estado final do banco de dados [Mohan et al. 1992, Mohan 1999].

A Figura 4.14 mostra um exemplo de CLR gerado por um *rollback*. A Figura 4.14 (a) representa as atualizações feitas por uma transação *T* e a Figura 4.14 (b) os registros armazenados no *log* através dessas atualizações. As ações *A1*, *A2*, *A3* e *A4* são atualizações normais e *L1*, *L2*, *L3* e *L4* são os respectivos registros de *log* dessas atualizações. *A3'* é uma ação de desfazer de *A3*. Assim, *L3'* é um registro de *log* de compensação que descreve *A3'*. Por exemplo, se *L3* representar uma operação de inserção de uma linha *R* em uma página *P* então *L3'* deve descrever a exclusão de *R* em *P*. Durante a recuperação, assumindo que a transação *T* não foi finalizada, as ações de *T* devem ser desfeitas, começando do registro *L4*. Quando o gerenciador de recuperação atingir o registro de *log* de compensação L3', ao invés de desfazer L3' e L3, ele pula para o registro L2 [Mohan et al. 1992, Mohan 1999].

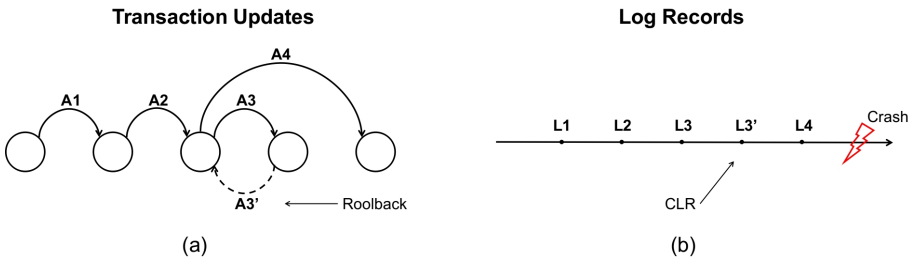


Figura 4.14. Registro de *log* de compensação gerado ao desfazer uma ação de transação [Mohan et al. 1992].

Os CLRs diminuem a quantidade de registros processados durante a recuperação após falhas, até em face a falhas sucessivas. Na ocorrência de tais eventos, os CLRs evitam o aninhamento de ações de desfazer, que podem causar um grande atraso no processo de recuperação. Em contraste, alguns sistemas, como DB2/MVS V1 [Cheng et al. 1984], NonStop SQL [Tandem Database Group 1987] e AS/400 [Clark and Corrigan 1989], não

usavam o conceito de CLRs e desfaziam a mesma ação de desfazer uma ou mais vezes em caso de falhas sucessivas [Mohan et al. 1992].

4.4.1.5. Granularidade fina

Sistemas antes de ARIES, como System R [Gray et al. 1981] e DB2/MVS V1 [Crus 1984], comumente implementavam o algoritmo UNDO/REDO para fins de tolerância a falhas [Bernstein et al. 1987]. Esse algoritmo executa a fase UNDO antes da REDO. Essa abordagem é chamada de paradigma de refazer seletivo (*paradigm of selective redo*). Nessa abordagem, a granularidade dos dados armazenados no *log* não pode ser maior que a dos dados manipulados no mecanismo de bloqueio (granularidade grossa). Por exemplo, se o SGBD armazena páginas no *log*, ele não deve manusear granularidade menor do que páginas nos bloqueios. Caso contrário, o banco de dados pode ficar inconsistente após uma recuperação. Isso acontece porque uma ação de desfazer durante o UNDO pode fazer o *log* perder o rastreamento para refazer uma transação durante o REDO [Mohan et al. 1992, Bernstein et al. 1987, Härder and Reuter 1983].

O algoritmo ARIES possui granularidade fina, ou seja, ele permite bloqueios de dados com granularidade menor que os dados do *log*. ARIES resolve o problema da granularidade grossa através do paradigma de repetição de história (*paradigm of repeating history*), ou seja, executando o passo REDO antes do UNDO [Mohan 1999, Mohan et al. 1992].

4.4.2. A família de algoritmos ARIES

Alguns algoritmos estendem ARIES acrescentando funcionalidades que o algoritmo original não implementa. São exemplos desses algoritmos: ARIES/NT (ARIES *for Nested Transactions*), ARIES-RRH (ARIES *with Restricted Repeating of History*), ARIES/IM (ARIES *for Index Management*), ARIES/KVL (ARIES *using Key-Value Locking*), ARIES/LHS (ARIES *for Linear Hashing with Separators*) e ARIES/CSA (ARIES *for the Client-Server Architecture*) [Mohan and Narang 1994, Mohan 1999].

ARIES/NT [Rothermel and Mohan 1989] acrescenta o uso de transações aninhadas. Uma transação aninhada (transação filha) é uma transação que inicia sua execução no escopo de execução de outra transação (transação pai). Esse algoritmo usa uma árvore para fazer o encadeamento dos registros de *log* das transações pai e filhas. A árvore é utilizada para percorrer os registros na ordem correta durante a recuperação do banco de dados.

ARIES/IM [Mohan and Levine 1992] e ARIES/KVL [Mohan 1990] tentam controlar a concorrência e recuperação de índices em árvores. Técnicas anteriores não consideravam a recuperação de índices com a granularidade fina de bloqueios. O principal objetivo desses algoritmos é resolver os problemas dos algoritmos anteriores de controle de concorrência de índices, como o algoritmo original de System R [Gray et al. 1981]. Por exemplo, ARIES/IM e KVL tentam evitar *deadlocks* entre transações durante o acesso aos índices. Um *deadlock* acontece quando duas ou mais transações entram em conflito por algum recurso e bloqueiam uma à outra permanentemente. Como consequência, a execução de nenhuma delas é possível.

ARIES/LHS [Mohan 1993] manuseia o controle de concorrência e recuperação de

índices em *hashes* dinâmicas. ARIES/LHS tenta evitar *deadlocks* de transações durante o acesso aos índices. Porém, esse algoritmo não foi implementado.

ARIES/CSA [Mohan and Narang 1994] foi projetado para trabalhar numa arquitetura cliente-servidor. Nesse algoritmo, as máquinas clientes fazem atualizações no banco de dados, produzem os registros de *log* dessas atualizações e enviam esses registros para o servidor armazená-los. O servidor é responsável por organizar os registros em seu arquivo de *log* local e gerenciar a recuperação do banco de dados. Esse algoritmo não foi implementado.

4.4.3. Recuperação através de *log* indexado

Existem técnicas modernas de recuperação de bancos de dados após falhas que utilizam *log* estruturado em forma de árvore, tais como: *Single-page repair* [Graefe and Kuno 2012], *Single-pass restore* [Sauer et al. 2015], *Instant restart* [Graefe et al. 2015, Graefe et al. 2016] e *Instant restore* [Sauer et al. 2017, Sauer 2017, Sauer 2019]. Ao invés do tradicional arquivo de *log* sequencial utilizado por ARIES, essas técnicas usam uma árvore B particionada [Graefe 2003] como arquivo de *log*. Existem também outras abordagens que utilizam outras estruturas de indexação, como árvore B^+, tabela *Hash* e árvore ART [Magalhães 2022, Magalhães et al. 2021a, Lee et al. 2022].

A árvore B particionada (Figura 4.15 (a)) armazena os registros em partições (Figura 4.15 (b)). Em cada partição, os registros de *log* são ordenados primariamente pelo ID da página que eles alteraram e secundariamente pela ordem das alterações. Cada partição é um arquivo que contém um índice que permite se deslocar aos registros de *log* de uma determinada página. No exemplo da Figura 4.15 (b), a partição possui registros de *log* que alteraram as páginas *A* e *B*. Cada chave de busca da árvore aponta para as partições que contêm registros de *log* que alteraram uma determinada página. Assim, é possível recuperar todos os registros de *log* de uma determinada página mediante uma busca na árvore [Graefe et al. 2015, Sauer et al. 2017, Sauer 2017].

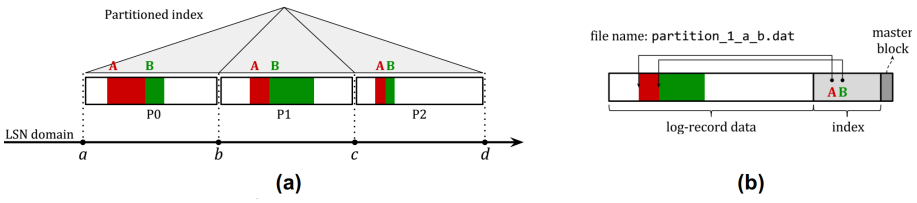


Figura 4.15. Árvore B particionada (a) e arquivo de partição (b) [Sauer 2017].

A recuperação do banco de dados via *log* indexado é realizada em paralelo à execução de transações, ou seja, o sistema não precisa esperar a recuperação completa para, só depois, começar a operação de novas transações. Essa abordagem é chamada de recuperação instantânea, pois os usuários/aplicações do SGBD têm a impressão de que o banco de dados foi recuperado imediatamente após uma falha [Graefe et al. 2015, Sauer et al. 2017, Sauer 2017, Magalhães et al. 2021a, Magalhães 2022].

A Figura 4.16 ilustra o esquema de recuperação instantânea de bancos de dados após uma falha de disco. Durante o processamento normal das transações, registros são armazenados no *log* indexado (árvore B particionada). Após uma falha, o processo de re-

cuperação copia páginas incrementalmente do *backup*. Enquanto um conjunto de páginas (seguimento) é copiado, os registros dessas páginas são buscados no *log* indexado. Em seguida, as ações dos registros são aplicadas em suas respectivas páginas. Assim que uma página é restaurada completamente, ela pode ser usada por novas transações. Caso uma nova transação precise de uma página que ainda não foi restaurada, essa página pode ser recuperada sob demanda por meio de buscas nos índices do *backup* e do *log* [Sauer et al. 2017, Sauer 2017, Sauer 2019].

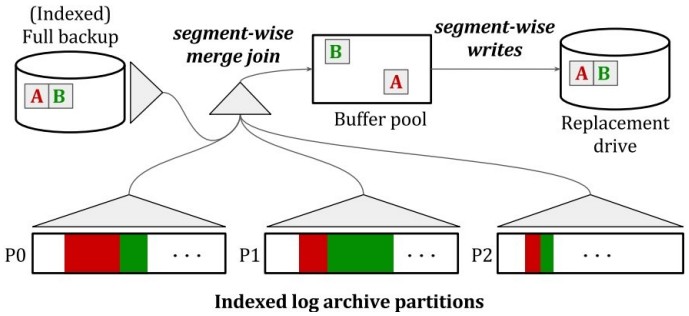


Figura 4.16. Esquema de recuperação instantânea após uma falha de disco [Sauer 2017].

4.5. Durabilidade e recuperação em bancos de dados em memória

Os bancos de dados em memória implementam técnicas de *logging*, *checkpoint* e recuperação para prover tolerância a falhas, como nos bancos em disco. Porém, quando examinadas em detalhes, a implementação dessas técnicas nesses dois sistemas se diferem muito. Por exemplo, a maioria dos bancos em memória evita protocolos estilo ARIES por questões de desempenho. Durabilidade e recuperação após falhas são as únicas razões para um SGBD em memória acessar o disco. Assim, além de prover tolerância a falhas, o mecanismo de recuperação também é responsável pela durabilidade dos SGBDs em memória [Jagadish et al. 1993, Gruenwald et al. 1996].

A Seção 4.4 forneceu os principais conceitos de recuperação de bancos de dados após falhas, focando em SGBDs em disco. Essa seção detalha como os SGBDs em memória implementam tolerância a falhas.

4.5.1. *Logging*

Registros de atualizações devem ser escritos para um arquivo de *log* em armazenamento estável devido à volatilidade da memória RAM. Um armazenamento estável é uma memória não volátil, onde dados podem persistir. Escrever registros em um arquivo de *log* é a maior fonte de sobrecarga para o processamento das transações nos SGBDs em memória devido às operações de E/S para a memória secundária. Assim, o mecanismo de *logging* é otimizado para interferir o mínimo possível no processamento das transações [Garcia-Molina and Salem 1992, Yao et al. 2016].

Os bancos de dados em memória comumente utilizam um *log* lógico que armazena registros com descrições das operações de atualização no banco de dados em alto nível, como a inserção de uma linha em uma tabela. Geralmente, registros lógicos são mais leves, pois armazenam menos itens de dados do que registros físicos utilizados pelos SGBDs em disco [Malviya et al. 2014, Wu et al. 2017, Zheng et al. 2014].

Para reduzir a quantidade de dados enviados para a memória secundária, o *log*

armazena registros REDO-*only* (apenas de refazer), ou seja, não são gravados registros de desfazer. Como consequência, transações devem gravar atualizações no *log* apenas durante o *commit*. Assim, caso o sistema falhe, não existirão atualizações de transações não finalizadas a serem desfeitas. Gravar registros apenas ao final de uma transação não é um problema para SGBDs em memória, pois eles são utilizados tipicamente em sistemas OLTP, que possuem transações de curta duração [Malviya et al. 2014, Wu et al. 2017].

Os SGBDs em memória ainda podem manter registros de UNDO temporariamente na memória principal para desfazer uma transação abortada. Após a transação ser finalizada ou desfeita (devido a um *abort*), os seus registros de desfazer são excluídos. Porém, muitos bancos de dados em memória não precisam desfazer transações, como nos sistemas multi versionados. Nesses sistemas, por exemplo, uma transação T_1 faz suas modificações em novas versões dos dados atuais do banco. Essas novas versões são vistas apenas pela transação T_1. Outras transações podem ver as versões de T_1 apenas após a finalização de T_1, ou seja, quando as versões de T_1 se tornam as versões atuais do banco de dados. Caso T_1 seja abortada, as suas versões são apenas descartas e, posteriormente, removidas por um coletor de lixo [Malviya et al. 2014, Wu et al. 2017, Zheng et al. 2014].

Para reduzir ainda mais o tráfego de dados para a memória secundária, alguns sistemas gravam registros de *log* ao nível de transação. Essa técnica é chamada de *logging* de transação ou de comando. Nessa técnica, cada transação deve ser um procedimento armazenado cadastrado previamente. Para cada transação executada, apenas um registro de *log* é gravado com o identificador do procedimento armazenado e seus parâmetros. Essa técnica leva a pouca sobrecarga no processamento das transações, pois apenas um registro é escrito no *log* para cada execução de transação, mesmo que essa transação possua muitas operações de escrita [Malviya et al. 2014, Wu et al. 2017].

Adicionalmente, muitos sistemas evitam escrever registros de atualização em índices no *log*. Assim, após uma falha, esses sistemas preferem reconstruir as estruturas de índice em paralelo ao processo de recuperação do banco de dados [Malviya et al. 2014, Wu et al. 2017, Zheng et al. 2014]. Esse minicurso não cobre recuperação de índices.

Os bancos de dados utilizam algumas técnicas para melhorar o desempenho de escrita de registros de *log* para a memória secundária, como *group commit* e *pre-commit*.

O *group commit* acumula registros de *log* produzidos por várias transações que executam no mesmo período de tempo para escrevê-los juntos em uma única operação de E/S. Essa técnica diminui o número de E/Ss para a memória secundária, uma vez que uma única E/S pode ser utilizada por várias transações [Hagmann 1987, DeWitt et al. 1984].

No *pre-commit*, uma transação libera os seus bloqueios assim que seus registros de *log* são enviados para a memória secundária, sem esperar a confirmação de escrita desses registros. Assim, a transação evita a sobrecarga de escrita para o *log*. Entretanto, o sistema pode perder as garantias de durabilidade em caso de falhas [DeWitt et al. 1984, Garcia-Molina and Salem 1992].

4.5.2. Checkpoint

Embora o *checkpoint fuzzy* (discutido na Seção 4.4.1) seja uma técnica que impõe pouca sobrecarga ao processamento de transações, ele não é adequado para SGBDs em memória.

Fuzzy depende de registros de *log* para refazer e desfazer transações. Porém, bancos de dados em memória não armazenam registros de UNDO. As técnicas de *checkpoint* em bancos em memória são muito diferentes das técnicas dos bancos em disco [Salem and Garcia-Molina 1989, Liedes and Wolski 2006, Ren et al. 2016].

Alguns SGBDs em memória materializam as operações lógicas do *log* em dados físicos armazenados em um arquivo de *checkpoint* na memória secundária. Assim, os registros de *log* anteriores a um *checkpoint* podem ser descartados, pois o arquivo de *checkpoint* funciona como um *backup* do banco de dados [Eich 1986, Garcia-Molina and Salem 1992, Diaconu et al. 2013, Stonebraker and Weisberg 2013].

A maioria dos bancos de dados em memória utiliza uma técnica de *checkpoint* consistente que produz um arquivo comumente chamando de *snapshot*. O *snapshot* é o *backup* do banco de dados em um dado instante de tempo. Porém, o sistema não precisa parar o processamento das transações para produzir um *snapshot* [Eich 1986, Garcia-Molina and Salem 1992, Diaconu et al. 2013, Stonebraker and Weisberg 2013].

Após um *checkpoint*, os registros de *log* anteriores ao *checkpoint* podem ser descartados. Como consequência, *checkpoints* reduzem o tempo de recuperação, pois menos registros de *log* precisam ser processados durante a recuperação. Além disso, carregar dados físicos para a memória é menos custoso do que executar operações lógicas, que requerem mais processamento do sistema. Além disso, geralmente um arquivo de *checkpoint* contém menos informação do que um arquivo de *log*. Por exemplo, sistemas OLTP fazem muitas modificações em poucas partes do banco de dados. Assim, um sistema OLTP tende a possuir muito mais registros no *log* do que dados no banco de dados [Eich 1986, Diaconu et al. 2013, Stonebraker and Weisberg 2013].

Foram propostos alguns algoritmos para produzir *snapshots* em SGBDs em memória, como Naive [Bronevetsky et al. 2006, Schroeder and Gibson 2007], Zigzag [Chen et al. 2011b], PingPong [Chen et al. 2011b], Hourglass [Li et al. 2018a, Li et al. 2018b] e Piggyback [Li et al. 2018a, Li et al. 2018b]. Porém, a maioria dos sistemas utiliza um algoritmo mais simples chamado de *Copy-on-Update* (COU).

O algoritmo de *snapshot* COU é executado durante a execução normal do sistema, ou seja, ele não para o processamento das transações. Esse algoritmo percorre todos os registros do banco de dados para copiá-los para um arquivo de *snapshot* na memória primária. Ele usa um *array* de *bits* para identificar registros inseridos, atualizados ou excluídos desde o início do processo de *checkpoint*. Assim, o algoritmo pode pular registros inseridos. Além disso, antes de uma atualização ou exclusão de registro, o conteúdo original desse registro é copiado para uma tabela sombra para que o *checkpoint* possa ler a versão antiga do registro. Durante o *checkpoint*, um processo serializa o *snapshot* da memória principal para um arquivo na memória secundária. Como desvantagem, essa técnica pode produzir uma sobrecarga de memória, pois ela pode potencialmente precisar de um espaço duas ou três vezes maior que o banco de dados [Chen et al. 2011b, Malviya et al. 2014].

4.5.3. Recuperação

O mecanismo de *logging* adotado pelos SGBDs em memória não produz inconsistências em face a falhas. Isso acontece porque os registros de REDO são gravados apenas na hora da confirmação das transações. Como consequência, não há a possibilidade de alguma transação não finalizada, devido a um *chash*, persistir dados inconsistentes. Além disso, registros de UNDO não são gravados no *log*. Assim, o processo de recuperação após uma falha de sistema em um SGBD em memória é, na verdade, um processo de carregar os dados da memória secundária para a memória principal [Magalhães et al. 2021b].

Sempre que acontece uma falha de sistema em um banco de dados em memória, o sistema para de funcionar e o conteúdo da memória principal é perdido. Como consequência, o banco de dados também é perdido. Assim que o sistema reinicia, o gerenciador de recuperação executa duas tarefas: (1) carregar o último *snapshot* da memória secundária para o banco de dados na memória principal e, em seguida, (2) reaplicar todas as operações dos registros de *log*. Após esse processo terminar, o sistema é recuperado para o seu último estado consistente de antes da falha e novas transações podem ser executadas [Eich 1986, Li and Eich 1993, Tang Yanjun and Luo Wen-hua 2010].

Não existe uma falha de disco em SGBDs em memória, visto que o banco de dados reside na memória principal. O dispositivo de armazenamento estável para onde são escritos os arquivos de *log* e *snapshot* pode falhar. Entretanto, isso não faz o sistema parar de funcionar, necessariamente. Porém, caso o sistema pare nesse evento, supondo que existem cópias do *log* e *checkpoint* em outros dispositivos, após a troca do dispositivo defeituoso, o *crash* pode ser tratado como uma falha de sistema [Eich 1986, Li and Eich 1993, Tang Yanjun and Luo Wen-hua 2010].

A Figura 4.17 ilustra a arquitetura básica de um banco de dados em memória. Durante o processamento normal das transações, o componente Logger copia registros de atualização no banco de dados para o arquivo de *log* na memória secundária. Além disso, periodicamente, o componente Checkpointer produz *snapshots* do banco de dados e os armazena na memória secundária. Após uma falha de sistema, todo o conteúdo do banco de dados é perdido. Assim que o SGBD reinicia, o componente Restorer copia o último *snapshot* para a memória principal e, em seguida, executa todas as operações dos registros de *log*. Novas transações podem ser executadas apenas após a recuperação completa do banco de dados [Magalhães et al. 2021b].

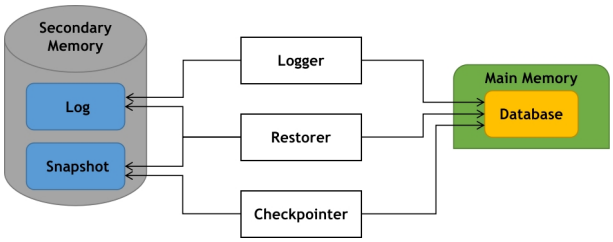


Figura 4.17. Arquitetura de um banco de dados em memória [Magalhães et al. 2021b].

4.6. Estratégias de recuperação de bancos de dados em memória

Existem várias abordagens para implementar um SGBD em memória, como discutido na Seção 4.3. Como consequência, as estratégias de recuperação adotadas por esses sistemas

são diversas. A Tabela 4.1 exibe as principais características das técnicas de tolerância a falhas implementadas por uma amostra representativa dos SGBDs em memória modernos. Os demais SGBDs em memória utilizam alguma combinação dessas técnicas.

Tabela 4.1. Estratégia de recuperação de SGBDs em memória e suas principais características.

SGBD	*Logging*	*Checkpoint*	Recuperação
Hekaton	operação	versões de dados	padrão
VoltDB	transação	*snapshot*	padrão
HyPer	transação	*snapshot*	padrão
SAP HANA	operação	*snapshot*	padrão
SiloR	valor	*snapshot "fuzzy"*	padrão
PACMAN	transação	-	paralela
Adaptive	transação/ estilo-ARIES	*snapshot*	paralela
FineLine	fisiológico	-	instantânea
HiEngine	valor	*snapshot*	"instantânea"
MM-Direct	operação	-	instantânea

Embora os SGBDs em memória utilizem várias técnicas diferentes para prover tolerância a falhas, as estratégias de recuperação são semelhantes. Assim, nesse trabalho, categorizamos essas estratégias em padrão, paralela e instantânea. As estratégias de recuperação foram categorizadas para simplificar a discussão delas. As seções seguintes detalham cada uma dessas estratégias.

4.6.1. Recuperação padrão

Hekaton, VoltDB, HyPer, SAP HANA e SiloR são exemplos de SGBDs em memória modernos que realizam tolerância a falhas como descrito na Seção 4.5, que é o método de recuperação utilizado pela maioria do SGBDs em memória. Contudo, se observados em detalhes, esse SGBDs diferem em como implementam suas estratégias de recuperação.

4.6.1.1. Hekaton

Hekaton é um banco de dados em memória incorporado ao Microsoft SQL Server. Assim, um banco de dados no SQL Server pode ter tabelas em disco e em memória. Uma transação pode atualizar os dois tipos de tabela. Contudo, uma transação que atualiza apenas tabelas em memória pode ser otimizada para acesso à memória. Hekaton utiliza armazenamento multi visionado e um MVCC otimista [Diaconu et al. 2013, Freedman et al. 2014, Larson et al. 2013]

Hekaton armazena versões de dados inseridos e atualizados em registros lógicos de REDO no arquivo de *log*. Para dados excluídos, apenas os IDs das versões desses dados são enviados para o *log*. *Checkpoints* são realizados periodicamente para fazer um mapeamento das versões no *log*. O *checkpoint* é armazenado em dois tipos de arquivo:

(*1*) *data* que contém as novas versões e (*2*) *delta* que contém informações das versões excluídas. Após uma falha, o arquivo *delta* é utilizado para filtrar quais versões no arquivo *data* devem ser carregadas na memória. Após a verificação completa do *checkpoint*, os registros de *log* após o *checkpoint* são reaplicados [Diaconu et al. 2013].

4.6.1.2. VoltDB

VoltDB é um SGBD em memória projetado a partir do banco de dados acadêmico H-Store [Kallman et al. 2008]. VoltDB armazena seus dados em partições com o objetivo de executar as transações serialmente. Além disso, o banco de dados pode ser distribuído e replicado em vários nós. Suas transações podem ser compiladas em procedimentos armazenados [Malviya et al. 2014, Stonebraker and Weisberg 2013].

VoltDB implementa *logging* de transações e *snapshots* para prover tolerância a falhas. Uma transação que executa em um único nó armazena seus registros de *log* apenas para o seu nó. Transações distribuídas executam em partições de mais de um nó. Nesse caso, um desses nós é escolhido para coordenar os demais. Apenas o coordenador armazena os registros de *log* gerados pela transação, como também as mensagens trocadas entre os nós [Malviya et al. 2014, Stonebraker and Weisberg 2013].

Após uma falha de sistema, o SGBD carrega o último *snapshot* na memória. Em seguida, cada um dos nós reaplica as ações de seus registros de *log*. Se necessário, o nó envia registros de *log* para outros nós os reaplicarem [Malviya et al. 2014, Stonebraker and Weisberg 2013].

4.6.1.3. HyPer

HyPer é um SGBD em memória capaz de suportar cargas de trabalho HTPA. Esse SGBD utiliza um esquema de memória virtual (Figura 4.18) para separar os dados entre os mais recentemente atualizados (para transações OLTP) e os mais imutáveis (para transações OLAP) [Faerber et al. 2017, Funke et al. 2014].

Inicialmente, transações OLTP e OLAP compartilham a mesma memória virtual. Quando um dado é atualizado, a nova versão desse dado é copiada para uma nova memória virtual acessada apenas por transações OLTP. As transações OLAP continuam acessando a versão antiga do dado na sessão de memória virtual em que foram criadas. Versões cujos dados não foram atualizados continuam sendo acessadas por transações OLTP e OLAP. As transações em HyPer executam suas operações sereialmente em partições [Faerber et al. 2017, Funke et al. 2014].

No exemplo da Figura 4.18, os objetos *a*, *a2* e *a3* representam versões antigas do objeto *a4* que é a versão atual de um dado. Apenas transações OLTP podem acessar a página do dado *a4*. As páginas de cada versão de dado são acessadas por suas correspondentes sessões OLAP [Funke et al. 2014].

HyPer implementa *logging* de transações e produz *snapshots*. Um *snapshot* é produzido como um *backup* do banco de dados através de uma sessão de memória virtual (ver Figura 4.18). Após um *crash*, o último *snapshot* é carregado para a memória e as

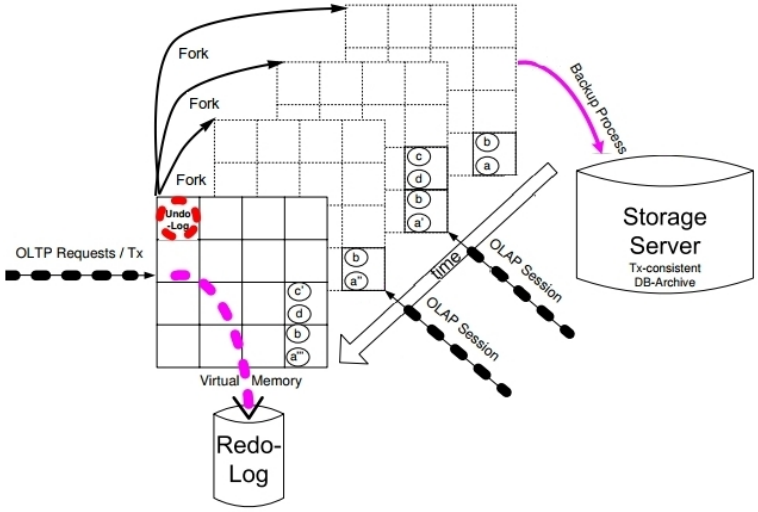


Figura 4.18. Esquema de memória virtual implementado por HyPer para suportar cargas de trabalho HTPA [Funke et al. 2014].

ações dos registros de *log* após o *chekcpoint* são executados novamente [Funke et al. 2014, Kemper and Neumann 2011, Mühe et al. 2011].

4.6.1.4. SAP HANA

O banco de dados SAP HANA é capaz de suportar cargas de trabalho HTPA através de uma estrutura de tabela unificada (Figura 4.19). A tabela unificada propaga tuplas para três representações de armazenamento: *L1-delta*, *L2-delta* e *Main store*. Em *L1-delta*, a tabela é orientada a linhas. *L2-delta* é uma estrutura de organização intermediária cuja tabela é orientada a colunas. *Main store* implementa tabelas no formato colunar com *dictionary encoding* (codificador de dicionário) altamente comprimido. O armazenamento de dados é multi versionado e o controle de concorrência é MVCC. [Färber et al. 2012, Färber et al. 2011, Faerber et al. 2017, Sikka et al. 2012].

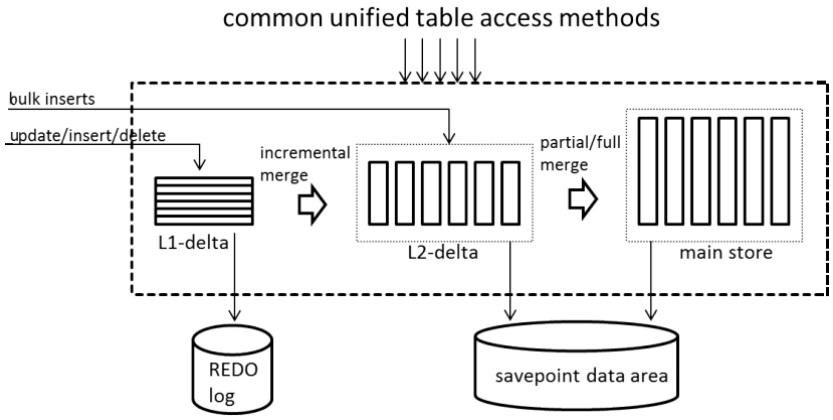


Figura 4.19. Esquema de tabela unificada implementada por SAP HANA [Sikka et al. 2012].

Apenas a parte *delta* do banco de dados manuseia operações OLTP: *L1-delta* para inserção, atualização e exclusão ou *L2-delta* para operações em lote. *Main store* é utilizado para operações OLAP. A medida que um dado é pouco atualizado, ele passa do armazenamento *L1-delta* para *L2-delta* e depois de *L2-delta* para *Main store* [Färber et al. 2012, Sikka et al. 2012].

SAP HANA implementa *logging* de operações e *savepoints* (*snapshots*). Transações que operam em *L1 e L2-delta* enviam registros lógicos de REDO para um arquivo de *log*. *Savepoints* são gerados apenas para as partes *L2-delta* e *Main store*. Após um *crash*, o sistema carrega o último *savepoint* na memória e refaz as ações do arquivo de *log* [Färber et al. 2012, Sikka et al. 2012].

4.6.1.5. SiloR

SiloR utiliza um esquema de épocas que implementa um número global *E* embutido no ID das transações. Uma *thread* é responsável por gerar o número *E* que é uma espécie de contador incrementado a cada 40 milissegundos. A abordagem de épocas impacta na maneira como SiloR implementa tolerância a falhas [Zheng et al. 2014].

SiloR armazena chaves e valores no *log* ao invés dos registros lógicos tipicamente implementados em sistemas em memória. O *logging* de valores permite uma recuperação em paralelo. Contudo, essa estratégia envia mais dados para o *log* aumentando o tempo de execução das transações. Os registros podem ser armazenados em vários arquivos de *log* e em qualquer ordem, uma vez que o número da época controla a ordem de execução das transações [Zheng et al. 2014].

O mecanismo de *checkpoint* divide o banco de dados em diferentes partes e atribui cada uma dessas partes a uma *thread* diferente. Essas *threads* copiam suas partes do banco de dados para arquivos de *checkpoint* diferentes. Como transações executam durante o processo de *checkpoint*, as *threads* podem não ver todas as modificações. Assim, SiloR implementa um *checkpoint* "*fuzzy*" [Zheng et al. 2014].

Após uma falha, *threads* carregam os arquivos de *checkpoint* na memória em paralelo. Em seguida, os arquivos de *log* são processados em paralelo. Ao final do processamento, cada dado é associado ao valor da sua última modificação. Então o *log* é reaplicado em ordem reversa. Essa abordagem permite que valores não sejam sobrescritos e, consequentemente, a CPU seja usada mais eficientemente [Zheng et al. 2014].

4.6.2. Recuperação em paralelo

Alguns sistemas em memória implementam uma estratégia de recuperação em paralelo com o objetivo de acelerá-la. Esses sistemas requerem que o arquivo de *log* grave registros ao nível de transação. Nessa caso, cada transação deve ser um procedimento armazenado previamente cadastrado, como descrito na Seção 4.5.1.

4.6.2.1. PACMAN

PACMAN é uma estratégia de recuperação implementada no banco de dados Peloton [Pavlo et al. 2017]. Sempre que um procedimento armazenado é compilado, PACMAN

divide-o em fatias (*slices*). Por exemplo, o procedimento *Transfer* da Figura 4.20 (a) possui 3 fatias. Em seguida, é construído um grafo de dependência local que identifica restrições de execução entre as fatias do procedimento armazenado. As restrições verificam se as operações conflitam em escrita ou devem obedecer uma determinada ordem. Assim, o grafo pode identificar possíveis oportunidades de execução em paralelo entre as fatias. A Figura 4.20 (b) apresenta o grafo de dependência do procedimento *Transfer* [Wu et al. 2017].

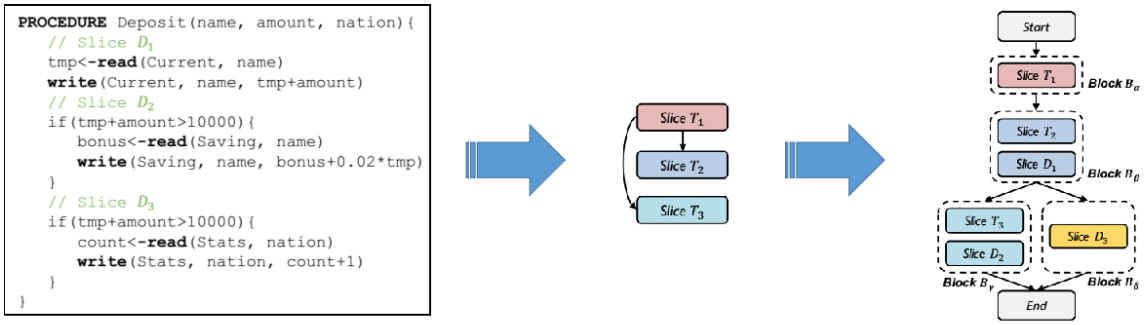


Figura 4.20. Procedimento armazenado (a), grafo de dependência local (b) e grafo de dependência global (c) [Wu et al. 2017].

O grafo de dependência local de cada procedimento armazenado é integrado a um grafo de dependência global que representa a ordem de execução de todas as fatias dos procedimentos armazenados. Através do grafo de dependência global, PACMAN é capaz produzir escalonamentos de execução das fatias dos procedimentos. Cada escalonamento pode executar em paralelo. Durante uma recuperação, dinamicamente, PACMAN tenta otimizar ainda mais o processo de recuperação. PACMAN permite mais execuções paralelas explorando a disponibilidade dos valores de parâmetros de procedimentos em tempo de execução e aplicando execução em *pipeline* [Wu et al. 2017].

4.6.2.2. Adaptive Logging

Adaptive Logging é uma estratégia de recuperação paralela implementada no SGBD H-Store. Essa estratégia é capaz de adaptar-se para melhorar o desempenho do processo de recuperação escolhendo quando é melhor utilizar *logging* de transações ou ARIES [Yao et al. 2016].

Após uma falha, antes de começar o processo de recuperação, o sistema percorre o arquivo de *log* para gerar um grafo de dependência entre as transações. Esse grafo relaciona transações que possuem escrita nos mesmos registros. Quando os registros de *log* estão sendo refeitos, transações sem relacionamento pode executar em paralelo. Caso contrário, as transações devem esperar até que suas dependências terminem de executar [Yao et al. 2016].

A dependência entre transações pode fazer poucas transações bloquearem muitas outras. Adaptive Logging pode identificar gargalos durante uma recuperação utilizando

um modelo de custo que usa um valor dado pelo usuário. Quando uma transação é identificada como gargalo, ela é materializada em *log* ARIES. Assim, transações dependentes do gargalo não precisam esperar muito [Yao et al. 2016].

4.6.3. Recuperação instantânea

Existem SGBDs em memória que usam estratégias de recuperação instantânea, ou seja, o sistema não precisa esperar por um longo período de tempo, após uma falha, para voltar a processar novas transações. Esses sistemas utilizam uma estrutura de índice no *log* ao invés do tradicional arquivo de sequencial [Magalhães et al. 2021a, Sauer 2019].

4.6.3.1. FineLine

FineLine implementa uma estratégia de recuperação instantânea cujo arquivo de *log* é uma árvore B particionada, descrita na Seção 4.4.3. Durante o processamento normal das transações, os registros de cada *group commit* são gravados em uma partição da árvore. Os registros de *log* armazenam páginas, ou seja, trata-se de um *log* físico [Sauer 2017, Sauer et al. 2018, Sauer 2019]. Essa abordagem prejudica o desempenho dos SGBDs em memória, uma vez que manusear um arquivo de índice é mais custoso do que um arquivo sequencial. Além disso, os registros de *log* físico são mais pesados do que os de *log* lógico.

Após uma falha, o sistema percorre o *log* indexado para recuperar páginas do banco de dados incrementalmente. Além disso, páginas podem ser buscadas na árvore para atender transações sob demanda. Para recuperar uma determinada página, o sistema deve percorrer múltiplas partições da árvore [Sauer 2017, Sauer et al. 2018, Sauer 2019].

FineLine não implementa *checkpoints*. Porém, o sistema mescla partições periodicamente para diminuir o tempo de recuperação de registros no *log*. Contudo, à medida que o arquivo de *log* aumenta, mais registros de *log* devem ser analisados durante uma recuperação após uma falha. Como consequência, quanto maior for *log*, mais lenta a recuperação será [Sauer 2017, Sauer et al. 2018, Sauer 2019].

4.6.3.2. HiEngine

HiEngine é um banco de dados desenvolvido pela Huawei [Huawei 2022]. Esse sistema armazena versões das tuplas do banco de dados em registros de um arquivo de *log* sequencial. Ele utiliza uma Árvore Radix Adaptativa (*Adaptive Radix Tree* - ART) como estrutura de indexação do *log*. A árvore funciona como um *array* indireto que aponta para os IDs das tuplas na memória ou para o endereço do último registro no *log* que modificou uma tupla. Essa abordagem permite a HiEngine carregar as tuplas do banco de dados na memória sob demanda. Contudo, manusear índices de *log* no processamento de transações pode degradar o desempenho do sistema. A Figura 4.21 ilustra a arquitetura de HiEngine [Lee et al. 2022, Ma et al. 2022].

Após uma falha, HiEngine reconstrói o *array* indireto na memória e as tuplas são carregadas na memória sob demanda. Periodicamente, HiEngine produz *snapshots* da

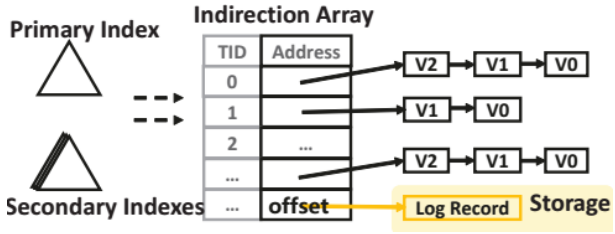


Figura 4.21. Arquitetura de HiEngine [Lee et al. 2022].

árvore. O *snapshot* armazena apenas o deslocamento para os registros no *log* ao invés dos registros. Dessa maneira, o tamanho do *snapshot* tende a ser pequeno e pode ser carregado na memória mais rapidamente. Segundo os autores, HiEngine pode recuperar o sistema após uma falha "instantaneamente"em aproximadamente 10 segundos e possui uma perda de desempenho aproximada de 5% a 11%. Adicionalmente, os arquivos de *log* e *snapshot* de HiEngine devem ser armazenados em um dispositivo de NVRAM, o que representa uma dependência dessa tecnologia [Lee et al. 2022].

4.6.3.3. MM-DIRECT

MM-DIRECT (*Main Memory Database Instant RECovery with Tuple consistent checkpoint* ou Recuperação instantânea de bancos de dados em memória com *checkpoint* consistente de tupla) é uma abordagem de recuperação implementada no banco de dados Redis [Redis 2020]. Essa abordagem permite a execução de novas transações imediatamente ao reinício do sistema após uma falha. Segundo os autores, o sistema possui um tempo de espera de aproximadamente 0,007 segundos após o seu reinício. MM-DIRECT requer um sistema simples contendo uma hierarquia de memória em dois níveis: (1) memória principal para o banco de dados e (2) memória persistente para os arquivos de *log* [Magalhães 2022, Magalhães et al. 2021a, Magalhães 2021].

Em MM-DIRECT, registros lógicos de REDO são enviados para um arquivo de *log* sequencial (Figura 4.22 (a)) durante o processamento normal das transações. Os registros são copiados do *log* sequencial para um arquivo de *log* indexado (Figura 4.22 (b)), que é uma árvore B^+. Esse processo é assíncrono ao processamento das transações, ou seja, as transações não precisam esperar pela escrita no *log* indexado para poder finalizar. Essa estratégia evita a degradação do desempenho do processamento de transações, uma vez que escrever registros para uma árvore B^+ é muito mais custoso do que escrevê-los para um arquivo sequencial. MM-DIRECT também suporta tabela *hash* como estrutura de *log* indexado [Magalhães 2022, Magalhães et al. 2021a, Magalhães 2021].

As chaves de busca da árvore B^+ são os IDs das tuplas do banco de dados. Cada nó da árvore contém uma lista dos registros de *log* que atualizaram uma determinada tupla. Os registros são ainda ordenados na lista pelo LSN. Dessa maneira, uma única busca na árvore pode encontrar todos os registros para recuperar uma tupla completamente na memória [Magalhães 2022, Magalhães et al. 2021a, Magalhães 2021].

Após uma falha, MM-DIRECT percorre o *log* indexado para recuperar o banco de dados. Cada visita a um nó da árvore recupera um tupla completamente na memória.

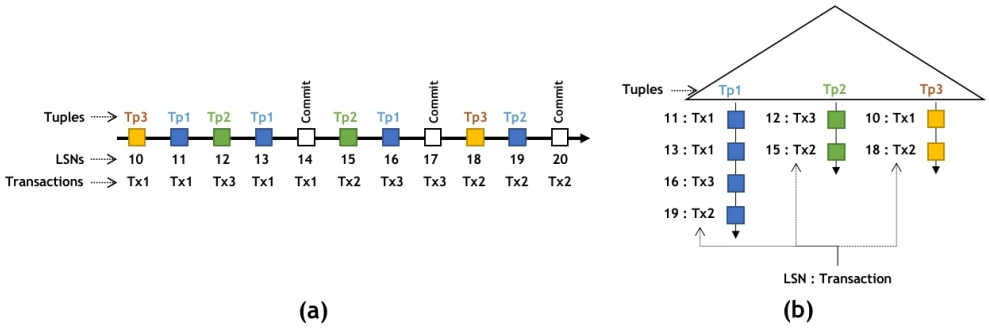


Figura 4.22. Log sequencial (a) e log indexado (b) [Magalhães et al. 2021a].

Assim que uma tupla é recuperada, ela pode ser acessada por novas transações. Contudo, caso uma transação precise de uma tupla que ainda não foi recuperada, a tupla pode ser recuperada sob demanda através de uma busca na árvore [Magalhães 2022, Magalhães et al. 2021a, Magalhães 2021].

MM-DIRECT implementa um mecanismo de *checkpoint* para reduzir o número de registros na árvore e, consequentemente, reduzir o tempo de recuperação após falha. Para cada nó da árvore, essa técnica substitui a lista de registros de *log* por um único registro cuja ação é equivalente à ação de todos os registros da lista. Assim, menos registros serão processados em caso de uma falha. Essa técnica ainda pode processar apenas os nós das tuplas mais frequentemente modificadas, diminuindo a sobrecarga do processo de *checkpoint*. Essa técnica de *checkpoint* é consistente, uma vez que suas modificações persistem caso o processo de *checkpoint* seja interrompido, por uma falha de sistema, por exemplo. [Magalhães 2022, Magalhães et al. 2021a, Magalhães 2021].

4.7. Principais desafios e direções futuras

Os bancos de dados em memória modernos tipicamente utilizam registros lógicos REDO-*only* com o objetivo de diminuir a quantidade de informação enviada para o *log* na memória secundária e, consequentemente, aumentar o desempenho no processamento das transações. Por outro lado, executar operações lógicas requer mais ciclos de CPU do que simplesmente copiar dados físicos para a memória. Como consequência, a recuperação em SGBDs em memória tende a ser mais lenta. Para acelerar a recuperação, alguns sistemas em memória têm utilizado alguma técnica de recuperação paralela [Magalhães et al. 2021b].

Algumas estratégias modernas de tolerância a falhas têm adotado técnicas radicalmente diferentes, como uso de estruturas de índice nos arquivos de *log* e *checkpoint*. Porém, manusear índices pode degradar o desempenho do sistema [Lee et al. 2022, Magalhães et al. 2021a, Sauer et al. 2018]. Além disso, criar *snapshots* de estruturas de índices não é tão simples quanto gerar *snapshots* do banco de dados ou do arquivo de *log* sequencial. HiEngine propôs alguns algoritmos de *snapshots* de *log* indexado, como ChainIndex, MirrorIndex e IACoW. Contudo, esses algoritmos ainda impõem alguma sobrecarga significativa ao processamento de transações [Lee et al. 2022].

Os sistemas de bancos de dados em memória têm tentado utilizar cada vez mais

o *hardware* moderno. Por exemplo, HiEngine usa NVRAM para acessar os arquivos de *log* e *snapshot* mais rapidamente [Lee et al. 2022]. HyPer usa RDMA para fazer *backups* e livrar o sistema dessa tarefa [Kemper and Neumann 2011]. Além de prover ganhos de desempenho aos sistemas em memória, as novas tecnologias têm fornecido oportunidades que não eram possíveis com *hardware* antigo. Porém, os sistemas se tornam dependentes dessas tecnologias.

4.8. Conclusões

Esse trabalho elucida as principais questões relacionadas a recuperação de bancos de dados, principalmente as relacionadas a bancos de dados em memória. Para isso, o trabalho provê uma visão geral das escolhas arquiteturais, de tecnologia e implementação de bancos de dados em memória e suas principais estratégias de recuperação após falhas de uma amostra representativa dos SGBDs em memória.

Referências

[Ailamaki et al. 1999] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Wood, D. A. (1999). Dbmss on a modern processor: Where does time go? In Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B., and Brodie, M. L., editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277. Morgan Kaufmann.

[Apache Software Foundation 2022] Apache Software Foundation (2022). Apache storm. Disponível em: "http://gridgain.com/". Acessado em: 19/12/2022.

[Arulraj and Pavlo 2017] Arulraj, J. and Pavlo, A. (2017). How to build a non-volatile memory database management system. In Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., and Suciu, D., editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1753–1758. ACM.

[Arulraj et al. 2015] Arulraj, J., Pavlo, A., and Dulloor, S. (2015). Let's talk about storage & recovery methods for non-volatile memory database systems. In Sellis, T. K., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 707–722. ACM.

[Arulraj et al. 2016a] Arulraj, J., Pavlo, A., and Menon, P. (2016a). Bridging the archipelago between row-stores and column-stores for hybrid workloads. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598. ACM.

[Arulraj et al. 2016b] Arulraj, J., Perron, M., and Pavlo, A. (2016b). Write-behind logging. *Proceedings of the VLDB Endowment*, 10(4):337–348.

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

[Binna et al. 2018] Binna, R., Zangerle, E., Pichl, M., Specht, G., and Leis, V. (2018). HOT: A height optimized trie index for main-memory database systems. In Das, G., Jermaine, C. M., and Bernstein, P. A., editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 521–534. ACM.

[Bishop et al. 2011] Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., and Velkov, R. (2011). OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42.

[Bitton et al. 1987] Bitton, D., Hanrahan, M., and Turbyfill, C. (1987). Performance of complex queries in main memory database systems. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 72–81. IEEE Computer Society.

[Bronevetsky et al. 2006] Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., and Stodghill, P. (2006). Recent advances in checkpoint/recovery systems. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE.

[Cha and Song 2004] Cha, S. K. and Song, C. (2004). P*time: Highly scalable OLTP DBMS for managing update-intensive stream workload. In Nascimento, M. A., Özsu, M. T., Kossmann, D., Miller, R. J., Blakeley, J. A., and Schiefer, K. B., editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 1033–1044. Morgan Kaufmann.

[Chen et al. 2001] Chen, S., Gibbons, P. B., and Mowry, T. C. (2001). Improving index performance through prefetching. In Mehrotra, S. and Sellis, T. K., editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 235–246. ACM.

[Chen et al. 2011a] Chen, S., Gibbons, P. B., and Nath, S. (2011a). Rethinking database algorithms for phase change memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 21–31. www.cidrdb.org.

[Chen et al. 2011b] Chen, S., Gibbons, P. B., and Nath, S. (2011b). Rethinking database algorithms for phase change memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 21–31. www.cidrdb.org.

[Cheng et al. 1984] Cheng, J. M., Looseley, C. R., Shibamiya, A., and Worthington, P. S. (1984). IBM database 2 performance: Design, implementation, and tuning. *IBM Systems Journal*, 23(2):189–210.

[Chhugani et al. 2008] Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y., Baransi, A., Kumar, S., and Dubey, P. (2008). Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324.

[Clark and Corrigan 1989] Clark, B. E. and Corrigan, M. J. (1989). Application system/400 performance characteristics. *IBM Systems Journal*, 28(3):407–423.

[Condie et al. 2010] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. (2010). Mapreduce online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 313–328. USENIX Association.

[Copeland and Khoshafian 1985] Copeland, G. P. and Khoshafian, S. (1985). A decomposition storage model. In Navathe, S. B., editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, pages 268–279. ACM Press.

[Crus 1984] Crus, R. A. (1984). Data recovery in IBM database 2. *IBM Systems Journal*, 23(2):178–188.

[DeWitt et al. 1984] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M., and Wood, D. A. (1984). Implementation techniques for main memory database systems. In Yormark, B., editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 1–8. ACM Press.

[Diaconu et al. 2013] Diaconu, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. (2013). Hekaton: SQL server's memory-optimized OLTP engine. In Ross, K. A., Srivastava, D., and Papadias, D., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM.

[Dragojevic et al. 2014] Dragojevic, A., Narayanan, D., Castro, M., and Hodson, O. (2014). Farm: Fast remote memory. In Mahajan, R. and Stoica, I., editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association.

[Eich 1986] Eich, M. H. (1986). Main memory database recovery. In *Proceedings of the Fall Joint Computer Conference, November 2-6, 1986, Dallas, Texas, USA*, pages 1226–1232. IEEE Computer Society.

[Eich 1987a] Eich, M. H. (1987a). A classification and comparison of main memory database recovery techniques. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 332–339. IEEE Computer Society.

[Eich 1987b] Eich, M. H. (1987b). MARS: the design of a main memory database machine. In Kitsuregawa, M. and Tanaka, H., editors, *Database Machines and Knowledge Base Machines, 5th International Workshop on Database Machines, Tokyo, Japan, 1987, Proceedings*, volume 43 of *The Kluwer International Series in Engineering and Computer Science*, pages 325–338. Kluwer.

[Elmasri and Navathe 2000] Elmasri, R. and Navathe, S. B. (2000). *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman.

[Faerber et al. 2017] Faerber, F., Kemper, A., Larson, P., Levandoski, J. J., Neumann, T., and Pavlo, A. (2017). Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130.

[Fan et al. 2013] Fan, B., Andersen, D. G., and Kaminsky, M. (2013). Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In Feamster, N. and Mogul, J. C., editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384. USENIX Association.

[Färber et al. 2011] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. (2011). SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51.

[Färber et al. 2012] Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., and Dees, J. (2012). The SAP HANA database – an architecture overview. *IEEE Database Engineering Bulletin*, 35(1):28–33.

[Freedman et al. 2014] Freedman, C., Ismert, E., and Larson, P. (2014). Compilation in the microsoft SQL server hekaton engine. *IEEE Database Engineering Bulletin*, 37(1):22–30.

[Funke et al. 2014] Funke, F., Kemper, A., Mühlbauer, T., Neumann, T., and Leis, V. (2014). Hyper beyond software: Exploiting modern hardware for main-memory database systems. *Datenbank-Spektrum*, 14(3):173–181.

[Garcia-Molina and Salem 1992] Garcia-Molina, H. and Salem, K. (1992). Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data*, 4(6):509–516.

[Garg et al. 2015] Garg, V., Singh, A., and Haritsa, J. R. (2015). On improving write performance in pcm databases. Technical report, Technical report, TR-2015-01, IISc.

[Graefe 2003] Graefe, G. (2003). Sorting and indexing with partitioned b-trees. In *First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org.

[Graefe et al. 2016] Graefe, G., Guy, W., and Sauer, C. (2016). *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

[Graefe and Kuno 2012] Graefe, G. and Kuno, H. A. (2012). Definition, detection, and recovery of single-page failures, a fourth class of database failures. *Proceedings of the VLDB Endowment*, 5(7):646–655.

[Graefe and McKenna 1993] Graefe, G. and McKenna, W. J. (1993). The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society.

[Graefe et al. 2015] Graefe, G., Sauer, C., Guy, W., and Härder, T. (2015). Instant recovery with write-ahead logging. *Datenbank-Spektrum*, 15(3):235–239.

[Gray et al. 1981] Gray, J., McJones, P. R., Blasgen, M. W., Lindsay, B. G., Lorie, R. A., Price, T. G., Putzolu, G. R., and Traiger, I. L. (1981). The recovery manager of the system R database manager. *ACM Computing Surveys (CSUR)*, 13(2):223–243.

[Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[GridGain Team 2022] GridGain Team (2022). Gridgain — extreme speed and scale for data-intensive apps | gridgain systems. Disponível em: "http://gridgain.com/". Acessado em: 19/12/2022.

[Gruenwald et al. 1996] Gruenwald, L., Huang, J., Dunham, Margaret H anA model of crash recovery in main memory databased Lin, J.-L., and Peltier, A. C. (1996). Recovery in main memory databases.

[Hagmann 1986] Hagmann, R. B. (1986). Crash recovery scheme for a memory-resident database system. *IEEE Computer Architecture Letters*, 35(9):839–843.

[Hagmann 1987] Hagmann, R. B. (1987). Reimplementing the cedar file system using logging and group commit. In Belady, L., editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 155–162. ACM.

[Hammond et al. 2004] Hammond, L., Wong, V., Chen, M. K., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. (2004). Transactional memory coherence and consistency. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 102–113. IEEE Computer Society.

[Härder and Reuter 1983] Härder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317.

[Harris 2016] Harris, R. (2016). Windows leaps into the nvm revolution. Disponível em: "https://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/". Acessado em: 22/07/2020.

[Harris et al. 2007] Harris, T., Cristal, A., Unsal, O. S., Ayguadé, E., Gagliardi, F., Smith, B., and Valero, M. (2007). Transactional memory: An overview. *IEEE Micro*, 27(3):8–29.

[Hazenberg and Hemminga 2011] Hazenberg, W. and Hemminga, S. (2011). Main memory database systems: Opportunities and pitfalls. *SC@ RUG 2011 proceedings*, page 113.

[Herlihy and Moss 1993] Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. In Smith, A. J., editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 289–300. ACM.

[Huawei 2022] Huawei (2022). Huawei - building a fully connected, intelligent world. Disponível em: "https://www.huawei.com". Acessado em: 22/04/2022.

[Hvasshovd et al. 1995] Hvasshovd, S., Torbjørnsen, Ø., Bratsberg, S. E., and Holager, P. (1995). The clustra telecom database: High availability, high throughput, and real-time response. In Dayal, U., Gray, P. M. D., and Nishio, S., editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 469–477. Morgan Kaufmann.

[Infinite Graph 2023] Infinite Graph (2023). Infinitegraph – limitless data possibilities. Disponível em: "https://infinitegraph.com". Acessado em: 12/05/2023.

[Jagadish et al. 1994] Jagadish, H. V., Lieuwen, D. F., Rastogi, R., Silberschatz, A., and Sudarshan, S. (1994). Dalí: A high performance main memory storage manager. In Bocca, J. B., Jarke, M., and Zaniolo, C., editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 48–59. Morgan Kaufmann.

[Jagadish et al. 1993] Jagadish, H. V., Silberschatz, A., and Sudarshan, S. (1993). Recovering from main-memory lapses. In Agrawal, R., Baker, S., and Bell, D. A., editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 391–404. Morgan Kaufmann.

[Kallman et al. 2008] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S. B., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. (2008). H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499.

[Karnagel et al. 2014] Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., and Lehner, W. (2014). Improving in-memory database index performance with intel® transactional synchronization extensions. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 476–487. IEEE Computer Society.

[Kemper and Neumann 2011] Kemper, A. and Neumann, T. (2011). Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In Abiteboul, S., Böhm, K., Koch, C., and Tan, K., editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society.

[Kim et al. 2010] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. (2010). FAST: fast architecture sensitive tree search on modern cpus and gpus. In Elmagarmid, A. K. and Agrawal, D., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 339–350. ACM.

[Lahiri et al. 2013] Lahiri, T., Neimat, M., and Folkman, S. (2013). Oracle timesten: An in-memory database for enterprise applications. *IEEE Database Engineering Bulletin*, 36(2):6–13.

[Larson and Levandoski 2016] Larson, P. and Levandoski, J. J. (2016). Modern main-memory database systems. *Proceedings of the VLDB Endowment*, 9(13):1609–1610.

[Larson et al. 2013] Larson, P., Zwilling, M., and Farlee, K. (2013). The hekaton memory-optimized OLTP engine. *IEEE Database Engineering Bulletin*, 36(2):34–40.

[Lee et al. 2013] Lee, J., Kwon, Y. S., Färber, F., Muehle, M., Lee, C., Bensberg, C., Lee, J., Lee, A. H., and Lehner, W. (2013). SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In Jensen, C. S., Jermaine, C. M., and Zhou, X., editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1165–1173. IEEE Computer Society.

[Lee et al. 2022] Lee, L., Xie, S., Ma, Y., and Chen, S. (2022). Index checkpoints for instant recovery in in-memory database systems. *Proc. VLDB Endow.*, 15(8):1671–1683.

[Lehman and Carey 1987] Lehman, T. J. and Carey, M. J. (1987). A recovery algorithm for A high-performance memory-resident database system. In Dayal, U. and Traiger, I. L., editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, pages 104–117. ACM Press.

[Lehman et al. 1992] Lehman, T. J., Shekita, E. J., and Cabrera, L. (1992). An evaluation of starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data*, 4(6):555–566.

[Leis et al. 2014a] Leis, V., Boncz, P. A., Kemper, A., and Neumann, T. (2014a). Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In Dyreson, C. E., Li, F., and Özsu, M. T., editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM.

[Leis et al. 2013] Leis, V., Kemper, A., and Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. In Jensen, C. S., Jermaine, C. M., and Zhou, X., editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society.

[Leis et al. 2014b] Leis, V., Kemper, A., and Neumann, T. (2014b). Exploiting hardware transactional memory in main-memory databases. In Cruz, I. F., Ferrari, E., Tao, Y., Bertino, E., and Trajcevski, G., editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 580–591. IEEE Computer Society.

[Levandoski et al. 2013] Levandoski, J. J., Lomet, D. B., and Sengupta, S. (2013). The bw-tree: A b-tree for new hardware platforms. In Jensen, C. S., Jermaine, C. M., and Zhou, X., editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 302–313. IEEE Computer Society.

[Li and Naughton 1988] Li, K. and Naughton, J. F. (1988). Multiprocessor main memory transaction processing. In Jajodia, S., Kim, W., and Silberschatz, A., editors, *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, USA, December 5-7, 1988*, pages 177–187. IEEE Computer Society.

[Li et al. 2018a] Li, L., Wang, G., Wu, G., and Yuan, Y. (2018a). Consistent snapshot algorithms for in-memory database systems: Experiments and analysis. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1284–1287. IEEE Computer Society.

[Li et al. 2018b] Li, L., Wang, G., Wu, G., Yuan, Y., Chen, L., and Lian, X. (2018b). A comparative study of consistent snapshot algorithms for main-memory database systems. *CoRR*, abs/1810.04915.

[Li and Eich 1993] Li, X. and Eich, M. H. (1993). Post-crash log processing for fuzzy checkpointing main memory databases. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 117–124. IEEE Computer Society.

[Li et al. 2013] Li, Y., Pandis, I., Müller, R., Raman, V., and Lohman, G. M. (2013). Numa-aware algorithms: the case of data shuffling. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org.

[Liedes and Wolski 2006] Liedes, A. and Wolski, A. (2006). SIREN: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In Liu, L., Reuter, A., Whang, K., and Zhang, J., editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 99. IEEE Computer Society.

[Lim et al. 2014] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. (2014). MICA: A holistic approach to fast in-memory key-value storage. In Mahajan, R. and Stoica, I., editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association.

[Lindström et al. 2013] Lindström, J., Raatikka, V., Ruuth, J., Soini, P., and Vakkila, K. (2013). IBM soliddb: In-memory database optimized for extreme speed and availability. *IEEE Database Engineering Bulletin*, 36(2):14–20.

[Lomet et al. 2012] Lomet, D. B., Fekete, A. D., Wang, R., and Ward, P. (2012). Multi-version concurrency via timestamp range conflict management. In Kementsietsidis, A. and Salles, M. A. V., editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 714–725. IEEE Computer Society.

[Low et al. 2012] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2012). Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078*.

[Ma et al. 2022] Ma, Y., Xie, S., Zhong, H., Lee, L., and Lv, K. (2022). Hiengine: How to architect a cloud-native memory-optimized database engine. In Ives, Z., Bonifati, A., and Abbadi, A. E., editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2177–2190. ACM.

[Maas et al. 2013] Maas, L. M., Kissinger, T., Habich, D., and Lehner, W. (2013). BUZZARD: a numa-aware in-memory indexing system. In Ross, K. A., Srivastava, D., and Papadias, D., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1285–1286. ACM.

[Magalhães 2021] Magalhães, A. (2021). Main memory databases instant recovery. In Bernstein, P. A. and Rabl, T., editors, *Proceedings of the VLDB 2021 PhD Workshop co-located with the 47th International Conference on Very Large Databases (VLDB 2021), Copenhagen, Denmark, August 16, 2021*, volume 2971 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Magalhães 2022] Magalhães, A. (2022). *Main memory database instant recovery*. PhD thesis, Federal University of Ceara, Brazil.

[Magalhaes et al. 2022] Magalhaes, A., Brayner, A., and Monteiro, J. M. (2022). Main memory database recovery strategies. In *Anais Estendidos do XXXVII Simpósio Brasileiro de Bancos de Dados*, pages 175–180. SBC.

[Magalhaes et al. 2023] Magalhaes, A., Brayner, A., and Monteiro, J. M. (2023). Main memory database recovery strategies. In *SIGMOD/PODS '23: Companion of the 2023 International Conference on Management of Data*, pages 31–35.

[Magalhães et al. 2021a] Magalhães, A., Brayner, A., Monteiro, J. M., and Moraes, G. (2021a). Indexed log file: Towards main memory database instant recovery. In Velegrakis, Y., Zeinalipour-Yazti, D., Chrysanthis, P. K., and Guerra, F., editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 355–360. OpenProceedings.org.

[Magalhães et al. 2017] Magalhães, A., Monteiro, J. M., and Brayner, A. (2017). Ajuste de performance em bancos de dados nosql. In *III Escola Reginal de Informática do Piauí , ERIPI 2017, Picos, PI, Brazil, 2017*.

[Magalhães et al. 2018a] Magalhães, A., Monteiro, J. M., and Brayner, A. (2018a). Gerenciamento e processamento de big data com bancos de dados em memória. In *I Jornada latino-americana de atualização em informática , JOLAI 2018, São Paulo, SP, Brazil, 2018*.

[Magalhães et al. 2018b] Magalhães, A., Monteiro, J. M., and Brayner, A. (2018b). Sistemas de gerenciamento de banco de dados em memória. In *XIV Simpósio Brasileiro de Sistemas de Informação , SBSI 2018, Caxias do Sul, RS, Brazil, 2018*.

[Magalhães et al. 2021b] Magalhães, A., Monteiro, J. M., and Brayner, A. (2021b). Main memory database recovery: A survey. *ACM Comput. Surv.*, 54(2):46:1–46:36.

[Makreshanski et al. 2015] Makreshanski, D., Levandoski, J. J., and Stutsman, R. (2015). To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proceedings of the VLDB Endowment*, 8(11):1298–1309.

[Malewicz et al. 2010] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In Elmagarmid, A. K. and Agrawal, D., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM.

[Malviya et al. 2014] Malviya, N., Weisberg, A., Madden, S., and Stonebraker, M. (2014). Rethinking main memory OLTP recovery. In Cruz, I. F., Ferrari, E., Tao, Y., Bertino, E., and Trajcevski, G., editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 604–615. IEEE Computer Society.

[Mao et al. 2012] Mao, Y., Kohler, E., and Morris, R. T. (2012). Cache craftiness for fast multicore key-value storage. In Felber, P., Bellosa, F., and Bos, H., editors, *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196. ACM.

[Menon et al. 2017] Menon, P., Pavlo, A., and Mowry, T. C. (2017). Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13.

[Mitchell et al. 2013] Mitchell, C., Geng, Y., and Li, J. (2013). Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In Birrell, A. and Sirer, E. G., editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 103–114. USENIX Association.

[Mohan 1990] Mohan, C. (1990). ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In McLeod, D., Sacks-Davis, R., and Schek, H., editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 392–405. Morgan Kaufmann.

[Mohan 1993] Mohan, C. (1993). ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 243–252. IEEE Computer Society.

[Mohan 1999] Mohan, C. (1999). Repeating history beyond ARIES. In Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B., and Brodie, M. L., editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 1–17. Morgan Kaufmann.

[Mohan et al. 1992] Mohan, C., Haderle, D., Lindsay, B. G., Pirahesh, H., and Schwarz, P. M. (1992). ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162.

[Mohan and Levine 1992] Mohan, C. and Levine, F. E. (1992). ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In Stonebraker, M., editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, pages 371–380. ACM Press.

[Mohan and Narang 1994] Mohan, C. and Narang, I. (1994). ARIES/CSA: A method for database recovery in client-server architectures. In Snodgrass, R. T. and Winslett, M., editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, pages 55–66. ACM Press.

[MongoDB Inc. 2022] MongoDB Inc. (2022). Mongodb: The developer data platform | mongodb. Disponível em: "http://www.mongodb.org". Acessado em: 19/12/2022.

[Mühe et al. 2011] Mühe, H., Kemper, A., and Neumann, T. (2011). How to efficiently snapshot transactional data: hardware or software controlled? In Harizopoulos, S. and Luo, Q., editors, *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, pages 17–26. ACM.

[MySQL 2020] MySQL (2020). Mysql. Disponível em: "http://www.mysql.com". Acessado em: 03/10/2020.

[MySQL 2022] MySQL (2022). Mysql. Disponível em: "http://www.mysql.com/". Acessado em: 20/12/2022.

[Neo4J 2023] Neo4J (2023). Neo4j graph data platform | graph database management system. Disponível em: "https://neo4j.com". Acessado em: 12/05/2023.

[Neumann 2011] Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550.

[Neumann et al. 2015] Neumann, T., Mühlbauer, T., and Kemper, A. (2015). Fast serializable multi-version concurrency control for main-memory database systems. In Sellis, T. K., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689. ACM.

[Neumeyer et al. 2010] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: distributed stream computing platform. In Fan, W., Hsu, W., Webb, G. I., Liu, B., Zhang, C., Gunopulos, D., and Wu, X., editors, *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 13 December 2010*, pages 170–177. IEEE Computer Society.

[Ooi et al. 2015] Ooi, B. C., Tan, K., Wang, S., Wang, W., Cai, Q., Chen, G., Gao, J., Luo, Z., Tung, A. K. H., Wang, Y., Xie, Z., Zhang, M., and Zheng, K. (2015). SINGA: A distributed deep learning platform. In Zhou, X., Smeaton, A. F., Tian, Q., Bulterman, D. C. A., Shen, H. T., Mayer-Patel, K., and Yan, S., editors, *Proceedings of the 23rd*

Annual ACM Conference on Multimedia Conference, MM '15, Brisbane, Australia, October 26 - 30, 2015, pages 685–688. ACM.

[Oracle 2020] Oracle (2020). Oracle | integrated cloud applications and platform services. Disponível em: "http://www.oracle.com". Acessado em: 04/12/2020.

[OrientDB 2023] OrientDB (2023). Home | orientdb community edition. Disponível em: "http://orientdb.org". Acessado em: 12/05/2023.

[Oukid et al. 2017] Oukid, I., Booss, D., Lespinasse, A., Lehner, W., Willhalm, T., and Gomes, G. (2017). Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177.

[Ousterhout et al. 2009] Ousterhout, J. K., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G. M., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. (2009). The case for ramclouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Oper. Syst. Rev.*, 43(4):92–105.

[Pandis et al. 2011] Pandis, I., Tözün, P., Johnson, R., and Ailamaki, A. (2011). PLP: page latch-free shared-everything OLTP. *Proceedings of the VLDB Endowment*, 4(10):610–621.

[Pavlo et al. 2017] Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D. V., Wang, Z., Wu, Y., Xian, R., and Zhang, T. (2017). Self-driving database management systems. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org.

[Porobic et al. 2014] Porobic, D., Liarou, E., Tözün, P., and Ailamaki, A. (2014). Atrapos: Adaptive transaction processing on hardware islands. In Cruz, I. F., Ferrari, E., Tao, Y., Bertino, E., and Trajcevski, G., editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 688–699. IEEE Computer Society.

[Porobic et al. 2012] Porobic, D., Pandis, I., Branco, M., Tözün, P., and Ailamaki, A. (2012). OLTP on hardware islands. *Proceedings of the VLDB Endowment*, 5(11):1447–1458.

[Ramakrishnan and Gehrke 2003] Ramakrishnan, R. and Gehrke, J. (2003). *Database management systems (3. ed.)*. McGraw-Hill.

[Rao and Ross 2000] Rao, J. and Ross, K. A. (2000). Making b^+-trees cache conscious in main memory. In Chen, W., Naughton, J. F., and Bernstein, P. A., editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486. ACM.

[Redis 2020] Redis (2020). Redis. Disponível em: "https://redis.io". Acessado em: 26/08/2020.

[Ren et al. 2016] Ren, K., Diamond, T., Abadi, D. J., and Thomson, A. (2016). Low-overhead asynchronous checkpointing in main-memory database systems. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1539–1551. ACM.

[Ren et al. 2012] Ren, K., Thomson, A., and Abadi, D. J. (2012). Lightweight locking for main memory database systems. *Proceedings of the VLDB Endowment*, 6(2):145–156.

[Rothermel and Mohan 1989] Rothermel, K. and Mohan, C. (1989). ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In Apers, P. M. G. and Wiederhold, G., editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 337–346. Morgan Kaufmann.

[Sadalage and Fowler 2019] Sadalage, P. J. and Fowler, M. (2019). *NoSQL essencial: um guia conciso para o mundo emergente da persistência poliglota*. Novatec Editora.

[Salem and Garcia-Molina 1989] Salem, K. and Garcia-Molina, H. (1989). Checkpointing memory-resident databases. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 452–462. IEEE Computer Society.

[Salem and Garcia-Molina 1990] Salem, K. and Garcia-Molina, H. (1990). System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data*, 2(1):161–172.

[Sauer 2017] Sauer, C. (2017). *Modern techniques for transaction-oriented database recovery*. PhD thesis, Kaiserslautern University of Technology, Germany.

[Sauer 2019] Sauer, C. (2019). Modern techniques for transaction-oriented database recovery. In Grust, T., Naumann, F., Böhm, A., Lehner, W., Härder, T., Rahm, E., Heuer, A., Klettke, M., and Meyer, H., editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme"(DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, volume P-289 of *LNI*, pages 487–496. Gesellschaft für Informatik, Bonn.

[Sauer et al. 2015] Sauer, C., Graefe, G., and Härder, T. (2015). Single-pass restore after a media failure. In Seidl, T., Ritter, N., Schöning, H., Sattler, K., Härder, T., Friedrich, S., and Wingerath, W., editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"(DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, volume P-241 of *LNI*, pages 217–236. GI.

[Sauer et al. 2017] Sauer, C., Graefe, G., and Härder, T. (2017). Instant restore after a media failure. In Kirikova, M., Nørvåg, K., and Papadopoulos, G. A., editors, *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, volume 10509 of *Lecture Notes in Computer Science*, pages 311–325. Springer.

[Sauer et al. 2018] Sauer, C., Graefe, G., and Härder, T. (2018). Fineline: log-structured transactional storage and recovery. *Proceedings of the VLDB Endowment*, 11(13):2249–2262.

[Schroeder and Gibson 2007] Schroeder, B. and Gibson, G. A. (2007). Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing.

[Schwalb et al. 2015] Schwalb, D., Berning, T., Faust, M., Dreseler, M., and Plattner, H. (2015). nvm malloc: Memory allocation for NVRAM. In Bordawekar, R., Lahiri, T., Gedik, B., and Lang, C. A., editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015*, pages 61–72.

[Shi et al. 2015] Shi, X., Chen, M., He, L., Xie, X., Lu, L., Jin, H., Chen, Y., and Wu, S. (2015). Mammoth: Gearing hadoop towards memory-intensive mapreduce applications. *IEEE Trans. Parallel Distributed Syst.*, 26(8):2300–2315.

[Sikka et al. 2012] Sikka, V., Färber, F., Lehner, W., Cha, S. K., Peh, T., and Bornhövd, C. (2012). Efficient transaction processing in SAP HANA database: the end of a column store myth. In Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L., and Fuxman, A., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 731–742. ACM.

[Silberschatz et al. 2020] Silberschatz, A., Korth, H. F., and Sudarshan, S. (2020). *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company.

[SingleStore 2022] SingleStore (2022). Singlestoredb is the real-time distributed sql database, designed for data-intensive applications. Disponível em: "https://www.singlestore.com". Acessado em: 20/12/2022.

[Stonebraker and Weisberg 2013] Stonebraker, M. and Weisberg, A. (2013). The voltdb main memory DBMS. *IEEE Database Engineering Bulletin*, 36(2):21–27.

[Strickland et al. 1982] Strickland, J. P., Uhrowczik, P. P., and Watts, V. L. (1982). IMS/VS: an evolving system. *IBM Systems Journal*, 21(3):490–510.

[Taft et al. 2014] Taft, R., Mansour, E., Serafini, M., Duggan, J., Elmore, A. J., Aboulnaga, A., Pavlo, A., and Stonebraker, M. (2014). E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proceedings of the VLDB Endowment*, 8(3):245–256.

[Tan et al. 2015] Tan, K., Cai, Q., Ooi, B. C., Wong, W., Yao, C., and Zhang, H. (2015). In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM Sigmod Record*, 44(2):35–40.

[Tandem Database Group 1987] Tandem Database Group (1987). Nonstop SQL: A distributed, high-performance, high-availability implementation of SQL. In Gawlick, D., Haynie, M. N., and Reuter, A., editors, *High Performance Transaction Systems, 2nd*

International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings, volume 359 of *Lecture Notes in Computer Science*, pages 60–104. Springer.

[Tang Yanjun and Luo Wen-hua 2010] Tang Yanjun and Luo Wen-hua (2010). A model of crash recovery in main memory database. In *2010 International Conference On Computer Design and Applications*, volume 5, pages V5–206–V5–207.

[Thomson et al. 2012] Thomson, A., Diamond, T., Weng, S., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: fast distributed transactions for partitioned database systems. In Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L., and Fuxman, A., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM.

[Tu et al. 2013] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. (2013). Speedy transactions in multicore in-memory databases. In Kaminsky, M. and Dahlin, M., editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 18–32. ACM.

[Tucker 2004] Tucker, A. B. (2004). *Computer science handbook*. CRC press.

[van Renen et al. 2018] van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., and Sato, M. (2018). Managing non-volatile memory in database systems. In Das, G., Jermaine, C. M., and Bernstein, P. A., editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1541–1555. ACM.

[VolDB 2022] VolDB (2022). Voldb active data. Disponível em: "http://www.voltdb.com". Acessado em: 20/12/2022.

[VoltDB Documentation 2022] VoltDB Documentation (2022). Volt active data documentation. Disponível em: "https://docs.voltdb.com". Acessado em: 27/12/2022.

[Wang et al. 2014] Wang, Z., Qian, H., Li, J., and Chen, H. (2014). Using restricted transactional memory to build a scalable in-memory database. In Bulterman, D. C. A., Bos, H., Rowstron, A. I. T., and Druschel, P., editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 26:1–26:15. ACM.

[Weikum and Vossen 2002] Weikum, G. and Vossen, G. (2002). *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann.

[Willhalm et al. 2013] Willhalm, T., Oukid, I., Müller, I., and Faerber, F. (2013). Vectorizing database column scans with complex predicates. In Bordawekar, R., Lang, C. A., and Gedik, B., editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013*, pages 1–12.

[Willhalm et al. 2009] Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. (2009). Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394.

[Wu et al. 2017] Wu, Y., Guo, W., Chan, C., and Tan, K. (2017). Fast failure recovery for main-memory dbmss on multicores. In Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., and Suciu, D., editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 267–281. ACM.

[Yao et al. 2016] Yao, C., Agrawal, D., Chen, G., Ooi, B. C., and Wu, S. (2016). Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1119–1134. ACM.

[Yoo et al. 2009] Yoo, R. M., Romano, A., and Kozyrakis, C. (2009). Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 198–207. IEEE Computer Society.

[Zaharia et al. 2013] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: fault-tolerant streaming computation at scale. In Kaminsky, M. and Dahlin, M., editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM.

[Zhang et al. 2015a] Zhang, H., Chen, G., Ooi, B. C., Tan, K., and Zhang, M. (2015a). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data*, 27(7):1920–1948.

[Zhang et al. 2015b] Zhang, H., Chen, G., Ooi, B. C., Wong, W., Wu, S., and Xia, Y. (2015b). "anti-caching-based elastic memory management for big data. In Gehrke, J., Lehner, W., Shim, K., Cha, S. K., and Lohman, G. M., editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1268–1279. IEEE Computer Society.

[Zhang et al. 2015c] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S. (2015c). Mojim: A reliable and highly-available non-volatile memory system. In Özturk, Ö., Ebcioglu, K., and Dwarkadas, S., editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 3–18. ACM.

[Zheng et al. 2014] Zheng, W., Tu, S., Kohler, E., and Liskov, B. (2014). Fast databases with fast durability and recovery through multicore parallelism. In Flinn, J. and Levy, H., editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 465–477. USENIX Association.