Capítulo 4

Sistemas de Gerenciamento de Banco de Dados em Memória

Arlino Magalhães, José Maria Monteiro e Ângelo Brayner

Abstract

The in-memory databases store their data directly in physical memory. This feature makes these databases provide a response time of orders of magnitude less than traditional disk-based systems. However, memory systems are more sensitive to failures, since main memory is volatile. Rather than attempting to optimize disk access, as in traditional databases, in-memory databases focus on optimizing CPU cycles and multicore processing. Differences between in-memory and disk databases affect almost every aspect of the database management, such us: data storage, indexing, concurrency control, durability and recovery techniques, and query processing and compilation. This work will discuss the main concepts and techniques that differentiate in-memory and disk resident databases. In addition, some in-memory database management systems will be introduced.

Resumo

Os bancos de dados em memória armazenam seus dados diretamente na memória física. Essa característica faz esses bancos possuírem um tempo de resposta de ordens de magnitude muito menor do que os sistemas tradicionais baseados em disco. Entretanto, os sistemas em memória são mais sensíveis a falhas, visto que, a memória principal é volátil. Ao invés de tentar otimizar acesso ao disco, como nos bancos de dados tradicionais, os bancos de dados em memória focam em otimizar ciclos de CPU e processamento multicore. As diferenças entre os bancos de dados em memória e os em disco afetam quase todos os aspectos de gerenciamento desses bancos, como: armazenamento, indexação, controle de concorrência, durabilidade e técnicas de recuperação, e processamento e compilação de consultas. Nesse trabalho serão discutidas os principais conceitos e técnicas que diferenciam os bancos residentes em memória dos bancos residentes em disco. Além disso, serão apresentados alguns sistemas de gerenciamento de bancos de dados em memória.

4.1. Introdução e motivação

Bancos de Dados em Memória (*Main Memory Databases - MMDBs* ou *In-Memory Databases - IMDBs*) são Sistemas de Gerenciamento de Bancos de Dados - SGBDs onde os dados estão armazenados diretamente na memória RAM em vez de permanecer em discos rígidos, como ocorre nos bancos de dados convencionais. Essa abordagem faz os bancos de dados em memória possuírem um tempo de resposta muito mais rápido do que os bancos de dados em disco. Apesar dos bancos de dados em memória serem apontados como uma tendência recente para solução de problemas, o desenvolvimento e pesquisa de bancos em memória começaram no início da década de 80. Entretanto, apenas na década de 90, com os avanças tecnológicos em *hardware*, os bancos de dados em memória começaram a refletir as tendências e arquiteturas utilizadas nos bancos de dados em memória mais modernos [Garcia-Molina and Salem 1992, Faerber et al. 2017].

Os SGBDs, tradicionalmente, são baseados em disco, ou seja, os dados são armazenados em disco rígido. Esses sistemas precisam copiar seus dados para memória RAM para que esses dados possam ser processados pela CPU e, se necessário, atualizações nos dados devem ser copiadas de volta para o disco. Nos bancos de dados em memória, os dados residem permanentemente na memória. Essa característica tem importantes implicações em como os bancos devem ser estruturados e acessados. A Figura 4.1 ilustra de forma básica as arquiteturas dos bancos de dados baseados em disco e dos baseados em memória. Os dados nos bancos em memória são acessados diretamente da memória RAM, eliminando o gargalho de I/O de acesso a disco. Entretanto, devido a volatilidade da memória RAM, os bancos em memoria são mais sensíveis a falhas (como falta de energia, por exemplo). Por prover tolerância a falhas, devem ser feitos *backups* (*logs* e *checkpoints*) periódicos dos dados para o disco. Os I/Os ao disco podem levar a possíveis *overheads* no sistema, mas para evitar isso os bancos de dados em memória utilizam estratégias para otimizar o acesso ao disco [Faerber et al. 2017, Zhang et al. 2015]. Na Seção 4.3.4 será brevemente discutido como implementar durabilidade e tolerância a falhas em bancos de dados.

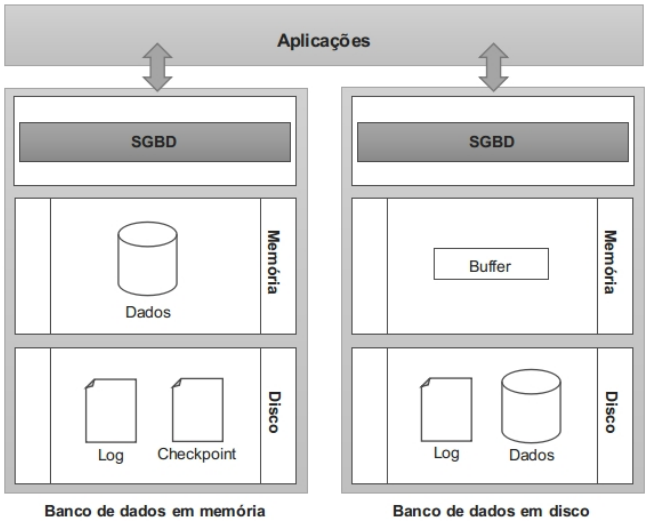


Figura 4.1. Arquitetura dos bancos de dados em memória *vesus* Arquitetura dos bacos de dados em discos.

A principal motivação para uso de bancos de dados em memória é sua baixa latência de acesso aos dados e possibilidades de análise em tempo real. Nas últimas décadas os *chips* de memória RAM tem dobrado em capacidade de armazenamento a cada 3 anos, enquanto que, seus preços tem caindo em um fator de 10 a cada 5 anos. Adicionalmente, já é comum as CPUs possuírem vários *cores* (até dezenas ou centenas). O processamento *multicore* tem sido largamente usado pelos bancos de dados em memória para ganhos de desempenho no processamento e acesso a dados. Como consequência, usar sistemas em memória principal para aplicações que tradicionalmente usam sistemas em disco tornou-se tecnicamente possível e economicamente viável [Zhang et al. 2015, Hazenberg and Hemminga 2011]. A memória principal claramente possui propriedades muito diferentes da memória secundária. Embora essas diferenças sejam bem conhecidas, segue um resumo delas [Garcia-Molina and Salem 1992]:

- O acesso à memória é de ordens de magnitude menor do que ao disco.
- A memória principal é volátil, enquanto que, o disco não é. Isso torna a memória mais vulnerável a fontes de *overhead* não existentes no disco, como durabilidade e tolerância a falhas, visto que, são necessários *backups* periódicos para o disco.
- Os discos são dispositivos de armazenamento orientados a blocos, equanto que, a memória não é. Essa característica permite uma representação mais flexível dos dados na memória e da forma mais adequada a melhorar o desempenho do sistema.
- O organização dos dados no disco é muito mais crítica do que na memória. Por exemplo, o acesso sequencial nos discos é mais rápido do que o acesso aleatório.

As diferenças entre as memórias principal e secundária se refletem nas diferenças entre os bancos de dados baseados em memória e os em disco. Os sistemas baseados em disco tentam otimizar o I/O, visto que, o acesso ao disco é um gargalho. Por esse motivo, esses sistemas utilizam a memória como uma *cache*. Além disso, caso a memória disponível não seja suficiente, técnicas de paginação são utilizadas para mover os dados em processamento na memória para o disco (e vice-versa), dando a impressão de que a memória utilizada é maior do que a disponível [Ramakrishnan and Gehrke 2002]. Os bancos de dados em memória acessam os dados diretamente da memória e os dados são organizados de forma a minimizar o *footprint* (quantidade de memória de trabalho necessária). Os sistemas em memória focam em otimizar o acesso à memória e ciclos de CPU [Hazenberg and Hemminga 2011]. A Tabela 4.1 sumariza as diferenças entre os bancos de dados em disco e os em memória.

Tabela 4.1. Diferenças entre os bancos de dados em disco e os em memória.

Banco de dados em disco	**Banco de dados em memória**
armazena dados em disco	armazena dados em memória
usa memória como *cache*	acessa a memória diretamente
assume que a memória é abundante	usa a memória disponível mais eficientemente
otimiza acesso ao disco	otimiza acesso à memória
mais ciclos de CPU	menos ciclos de CPU

As diferenças entre bancos em disco e os em memória afetam quase todos os aspectos de gerenciamento desse dois tipos de bancos de dados. Na Seção 4.3 serão descritos brevemente as principais detalhes relacionadas a *design* e escolhas arquitetônicas. Entretanto, antes disso, é importante salientar algumas questões que são comumente perguntadas sobre bancos de dados em memória:

1. **Um banco de dados inteiro pode ser armazenado na memória?**
   Sim! Alguns bancos de dados tem um tamanho limitado e/ou sua taxa de crescimento é muito menor do que a taxa de crescimento atual de armazenamento das memórias RAMs. Por exemplo: o tamanho de um banco de dados pode ser proporcional a quantidade de empregados ou clientes de uma empresa e essas informações podem não precisar de muitos *bytes* para serem armazenadas. Além disso, existem dados raramente acessados. De acordo com a regra 80-20, oitenta por cento dos acessos a bancos de dados são a apenas vinte porcento dos dados. Assim, os dados podem ser divididos de acordo com a frequência de acesso, como por exemplo em: frequentemente acessados (*hot*) e raramente acessados (*cold*). Apenas os dados *hot* precisam ser carregados na memória, enquanto que, os *cold* podem permanecer em disco até serem necessários [Garcia-Molina and Salem 1992, Gruenwald and Eich 1994]. Adicionalmente, caso realmente a base de dados seja muito maior do que a quantidade de memória disponível, a base pode ser distribuída entre as memórias dos vários nós de um banco de dados em memória distribuído [Kemper and Neumann 2011, Färber et al. 2012].

2. **Utilizar um banco de dados baseado em disco com memória RAM suficiente para armazenar a base de dados inteira é equivalente a utilizar um banco de dados baseado em memória?**
   Não! Possuir memória RAM suficiente para armazenar toda a base de um banco de dados baseado em disco pode trazer ganhos de desempenho, visto que, minimiza a quantidade de acessos ao disco. Entretanto, eventuais atualizações nos dados devem ser descarregadas para o disco, o que leva a I/Os. Além disso, mesmo com um *buffer* grande, as estruturas de dados ainda continuam por tentar otimizar o acesso ao disco. Nos bancos de dados em memória, os dados residem permanentemente na memória e as estratégias de acesso focam em otimizar a memória [Hazenberg and Hemminga 2011, Faerber et al. 2017]. A Seção 4.3.1 trata melhor da organização e *layout* dos dados.

3. **O bancos de dados em memória podem perder todos os seus dados após um desligamento de energia?**
   Não! Os bancos de dados em memória implementam estratégias de durabilidade e recuperação após falhas. Como a memória principal é volátil, as informações contidas no banco de dados devem ser movidas periodicamente para a memória secundária. I/Os podem implicar em *overheads* no sistema. Entretanto, os bancos de dados em memória utilizam estratégias para otimizar os *commit*, *log* e *checkpoint* com o objetivo de prover durabilidade e recuperação após falhas. O *commit* é a efetivação de atualizações de uma transação no bancos de dados. O *log* contêm informações de atualizações feitas no banco de dados. O *checkpoint*, em bancos de dados em memória, é considerado como o *backup* mais recentemente atualizado da

base de dados. *Commit*, *log* e *checkpoint* precisam ser descarregados para o disco e, por isso, devem ser realizados de maneira a não comprometer o funcionamento normal do sistema [Mohan and Levine 1992, DeWitt et al. 1984]. Durabilidade e tolerância a falhas serão melhor explicadas na Seção 4.3.4.

4. **Memória RAM não volátil pode ser utilizada em bancos de dados em memória no lugar da memória RAM convencional?**
   Sim! Memória RAM não volátil (NVRAM) é uma tecnologia que captura as principais vantagens da memória RAM (alto desempenho) e do disco rígido (persistência). Entretanto, o uso de NVRAM para armazenar grandes quantidades de dados ainda é economicamente proibitivo. Mas essa tecnologia promete ser promissora em um futuro próximo [Zhang et al. 2015]. Segundo Jim Gray *"Memory is the new disk, disk is the new tape"*, visionando que a memória principal iria substituir o disco rígido e este último seria utilizado apenas para *backup* [Robbins 2008]. Na Seção 4.2.4, a memória NVRAM será abordada com mais detalhes.

5. **Que tipo de aplicações tipicamente empregam bancos de dados em memória?**
   Bancos de dados em memória são utilizados comumente em aplicações que demanda altas velocidades de acesso, armazenamento e manipulação de dados. Alguns exemplos são sistemas de roteamento de IPs em redes, controle industrial, roteamento de ligações telefônicas, mercados financeiros, *e-comerce*, redes sociais e etc. Vários sistemas embarcados, como *MP3 players*, utilizam bancos de dados em memória. Recentemente, o fenômeno *Big Data* tem impulsionado pesquisas no desenvolvimento de sistemas com serviços de latência muito baixa (milissegundos) e análise de dados em tempo real. Os sistemas baseados em disco existentes não conseguem atingir esse tempo de reposta devido a alta latência dos discos rígidos. Para atingir estritamente requisitos de análise de grandes quantidades de dados em tempo real e atender requisições em milissegundos, os sistemas de bancos de dados em memória tem sido utilizados [Zhang et al. 2015].

4.2. Tecnologias *core* para bancos de dados em memória

Nessa seção serão introduzidos alguns conceitos e técnicas que são a base para desempenho de bancos de dados em memória, incluindo hierarquia da memória, acesso não uniforme a memória, memória transacional e memória RAM não volátil.

4.2.1. Hierarquia da memória

A hierarquia da memória é definida em termos de latência com os seguintes componentes: registrador, *cache* (L1, L2 e L3), memória principal e disco (disco rígido, memória *flash* e SSD). A Figura 4.2 ilustra a hierarquia da memória mostrando do componente mais rápido para o mais lento, seguindo a ordem de cima para baixo. Nas arquiteturas modernas, a CPU não pode processar dados que não estejam nos registradores. Assim, o dado deve ser transmitido através de cada uma das camadas de memória até atingir o registrador. Consequentemente, cada camada funciona como uma *cache* da camada adjacente inferior para reduzir a latência de acessos repetidos. O desempenho de sistemas de uso intensivo de dados depende muito da hierarquia da memória. Entretanto, esses sistemas também

precisam de otimizações para atingir localização espacial e temporal. Localização espacial assume que dados adjacentes são mais comumente acessados juntos. A localização temporal refere-se a observação de que um dado acessado será novamente acessado em um futuro próximo [Zhang et al. 2015].

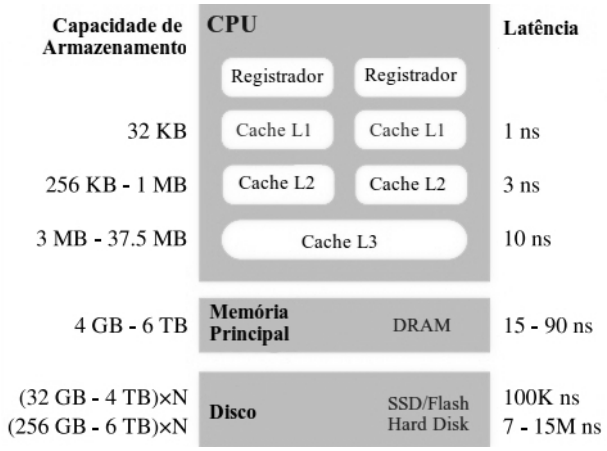


Figura 4.2. Hierarquia da memória
Fonte: [Zhang et al. 2015]

4.2.2. Acesso não uniforme a memória

O Acesso não Uniforme a Memória (*Non-Uniform Memory Access* - NUMA) é uma arquitetura para projeto de memória principal de computadores multiprocessados. Nessa arquitetura, o acesso à memória executado pelos processadores é não uniforme, ou seja, a latência de uma operação na memória depende da localização relativa do processador que está executando essa operação. Cada processador, em um sistema NUMA, possui uma memória local cuja latência é mínima, mas o processador também pode fazer acessos remotos a outras memórias cuja latência é maior [Li et al. 2013, Zhang et al. 2015]. A Figura 4.3 esboça uma arquitetura NUMA com N nós.

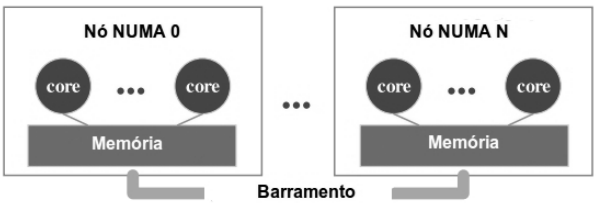


Figura 4.3. Tecnologia NUMA
Fonte: [Zhang et al. 2015]

A motivação para empregar a arquitetura NUMA é melhorar a largura de banda da memória e o tamanho total de memória que pode ser implementado em um nó do servidor. No contexto de bancos de dados, as pesquisas em NUMA podem ser divididas em (1) particionamento dos dados de maneira que o acesso remoto em NUMA seja minimizado [Maas et al. 2013, Leis et al. 2014a]; (2) gerenciamento dos efeitos de NUMA em cargas de trabalho sensíveis a latência, como cargas de transações OLTP

[Porobic et al. 2012, Porobic et al. 2014]; e (3) eficiência de transferência de dados através de domínios NUMA [Li et al. 2013].

4.2.3. Memória transacional

Memória Transacional (*Transactional Memory* - TM) é um mecanismo para controle de concorrência a dados compartilhados em sistemas concorrentes, análogo ao controle de concorrência em transações de banco de dados. Existem dois tipos de memória transacional: Memória Transacional em *Software* (*Software Transactional Memory* - STM) [Saha et al. 2006, Shavit and Touitou 1997], que implementa o sistema de execução transacional em *software*, utilizando bibliotecas e compiladores, por exemplo; e Memória Transacional em *Hardware* (*Hardware Transactional Memory* - HTM) [Knight 1986, Zilles and Rajwar 2007], que implementa o sistema de execução transacional por meio de suporte arquitetural, modificando *cache*, barramento ou protocolos de coerência, por exemplo. Uma terceira possibilidade é chamada de Memória Transacional Híbrida (*Hybrid Transactional Memory* - HyTM), que visa obter melhor desempenho utilizando características de STM e HTM [Lintzmayer et al.].

HTMs e STMs são responsáveis por: identificar acessos a memória dentro de transações, gerenciar os conjuntos de leitura e escrita da transação, detectar e resolver conflitos, gerenciar o estado dos registradores da arquitetura, efetivar e abortar transações. STM oferece as vantagens de (1) permitir a implementação de uma grande variedade de algoritmos sofisticados; (2) possuir modificação de *software* fácil e barata; e (3) poder ser integradas mais facilmente com sistemas existentes. Embora STMs possam ser bem flexíveis, elas causam um *overhead* significante ao sistema, inviabilizando sua aplicação prática. HTM tem atraído mais atenção por (1) oferecer quase nenhum *overhead* a execução das transações; (2) e por ser menos invasiva nos sistemas atuais. Como desvantagem, HTM não suporta transações grandes [Lintzmayer et al. , Cascaval et al. 2008]. Atualmente já existem implementações de HTM nos processadores modernos que estão sendo utilizados nos bancos de dados em memória, como o Haswell da Intel [Leis et al. 2014b].

4.2.4. Memória RAM não volátil

A Memória RAM não Volátil (*Non-Volatile Random Access Memory* - NVRAM) é uma tecnologia emergente com perspectivas de persistência com alta velocidade e grande capacidade de armazenamento. Existem algumas implementações de NVRAM; como *Phase-Change Memory* - PCM [Raoux et al. 2008], Memristors [Strukov et al. 2008] e *Spin-Transfer Torque Magnetic RAM* - STT-MRAM [Driskill-Smith 2010]; que podem prover um desempenho semelhante a DRAM, mas com persistência. PCM é apenas duas vezes mais lenta que a DRAM, e STT-MRAM e Memristor podem atingir latência maior que a DRAM. Embora NVRAM esteja atualmente disponível apenas em pequenas quantidades, especula-se que na próxima década haverão memórias PCM de 1TB e Memristors de 30TB ao preço dos discos rígidos atuais [Zhang et al. 2015]. Também existem vários trabalhos explorando o uso de NVRAM para armazenamento em bancos de dados, como: [Arulraj et al. 2015], [Chatzistergiou et al. 2015] e [Oukid et al. 2014]. Esses trabalhos utilizam simuladores de memória NVRAM para fazer seus experimentos, como o Intel Lab's NVM Hardware Emulator [Dulloor et al. 2014].

4.3. *Design* e escolhas arquitetônicas

Nessa seção serão discutidos brevemente aspectos chaves que fazem os bandos de dados em memória possuírem alto desempenho. Esses aspectos influenciam a arquitetura e técnicas de implementação do gerenciamento do bancos de dado. Para cada questão de *desing* e escolha arquitetônica, primeiro será mostrado como elas são implementadas em bancos de dados em disco para, em seguida, ser explicado como elas são implementadas nos bancos de dados em memória.

4.3.1. Organização dos dados e *layout*

Quando uma requisição é recebida por um banco de dados em disco, o gerenciador de *buffer* deve checar se a página do dado requisitado está no *buffer* (porção da memória destinada aos dados) ou não. Se essa página não estiver na memória, ela deve ser carregada do bloco no disco para a memória. Uma página pode conter vários registros de dados e possui uma representação semelhante ao bloco do disco para evitar traduções entre as duas representações. Se a página já estiver carregada na memória, muitos gerenciadores de *buffer* implementam acesso indireto ao dado: (1) encontrar a página do dado requisitado no *buffer*, através de uma tabela *hash*, por exemplo; e (2) calcular o descolamento (*offset*) necessário dentro da página para acessar o registro do dado [Ramakrishnan and Gehrke 2002]. A Figura 4.4 esquematiza o funcionamento do gerenciador de *buffer*.

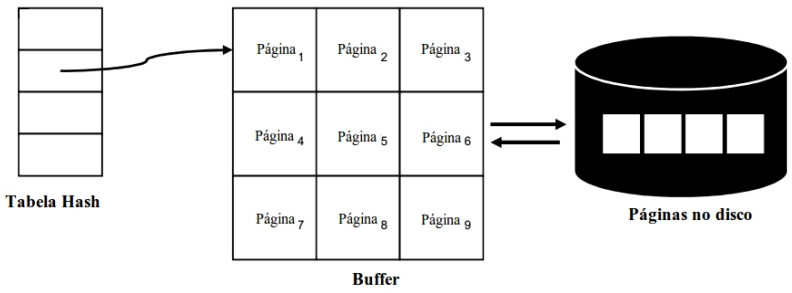


Figura 4.4. Gerenciador de *buffer* em um SGBD baseado em disco.
Fonte: [Faerber et al. 2017]

Os bancos de dados em memória não são orientados a blocos e suas representações são implementadas de forma a levar ao melhor desempenho no sistema. São exemplos de *layouts* de dados usados nos sistemas em memória modernos: particionado, multi-versionado e linha/coluna. Como os dados residem permanentemente na memória, não há a necessidade carregá-los do disco para o *buffer*. Comumente, os registros são acessados diretamente através de ponteiros. Acessar os dados diretamente na memória leva a menos ciclos de CPU, evitando o *overhead* do acesso indireto. As estratégias de acesso focam em otimizar ciclos de CPU e paralelismo *multi-core* [Hazenberg and Hemminga 2011, Faerber et al. 2017].

4.3.2. Indexação

Índices são estruturas de dados que permitem rápido acesso a dados, sem a necessidade de percorrer uma tabela inteira. A principal meta dos índices, nos sistemas baseados em disco, é minimizar o acesso ao disco, mapeando dados do banco de dados. Para

isso, tipicamente, os bancos de dados utilizam índices em árvores B+. Uma alternativa é o índice com tabela *hash*, muito eficiente em buscas com um valor específico. Sistemas em disco também utilizam índices clusterizados, que ordenam os registros em uma ordem específica no disco. Esse tipo de índice é muito eficiente para buscas usando faixas de valores. Entretanto, cada tabela só pode ter um índice clusterizado (índice primário), enquanto que, pode ter vários dos outros tipos de índices (índices secundários) [Ramakrishnan and Gehrke 2002, Elmasri and Navathe 2010].

Nos sistemas em memória, a principal meta dos índices é reduzir o tempo computacional e o uso da memória ao máximo possível. Os índices armazenam ponteiros para os registros em vez de *IDs* ou chaves primárias, como é feito nos índices para bancos em disco. A Figura 4.5 ilustra as diferenças entre os índices de sistemas em discos e em memória. Como a velocidade das CPUs modernas é superior a da memória RAM, a memória torna-se um gargalho em sistemas em memória. Assim, para resolver esse problema, técnicas de otimização do uso da *cache* são implementadas em bancos de dados em memória, como índices de árvores CSB+ [Rao and Ross 2000] e Pb+ [Chen et al. 2001], por exemplo. Os índices são estrutura frequentemente acessadas e atualizadas, o que pode comprometer a concorrência e o paralelismo. Os sistemas em memória utilizam técnicas, como PLP [Pandis et al. 2011] e árvores Mass [Mao et al. 2012], para atingir altos níveis de concorrência e paralelismo.

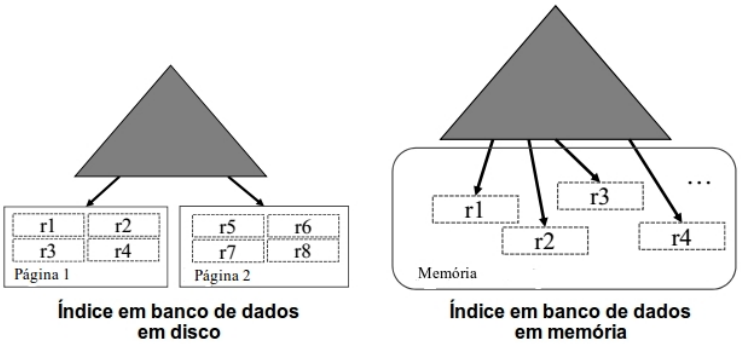


Figura 4.5. Estruturas de índices nos bancos de dados em disco e em memória. Fonte: [Faerber et al. 2017]

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. Os bancos de dados relacionais em disco, tipicamente, utilizam alguma forma de bloqueio em duas fazes (*Two-Phase Locking* - 2PL) para garantir o controle de concorrência. No protocolo 2PL, antes de começar a executar, uma transação deve adquirir bloqueios a todos os dados que necessita. Em uma segunda faze, a transação deve liberar os bloqueios adquiridos, após não precisar mais dos dados. Uma transações que necessite alterar dados bloqueados por uma outra transação deve esperar até a liberação desses bloqueios [Ramakrishnan and Gehrke 2002, Elmasri and Navathe 2010].

O gerenciador de bloqueio realiza as operações de bloqueio e desbloqueio atomicamente, ou seja, nenhuma outra operação no banco de dados é permitida enquanto

uma delas está em execução. Além disso, sempre que uma transação precisa ler/escrever em um dado, o gerenciador de bloqueio deve verificar se o dado já está bloqueado por outra transação. Essas operações realizadas pelo gerenciador de bloqueio podem ser toleradas em bancos de dados em disco devido ao gargalho de acesso ao disco, mas como o acesso à memória é muito mais rápido do que ao disco, em sistemas em memória, bloqueios tornam-se muito custosos e acrescentam mais ciclos de CPU. Por esse motivo, os sistemas em memória evitam protocolos de bloqueio e preferem executar as transações serialmente ou adotar um modelo de controle de concorrência multi-versionado (*Multi-Version Concurrency Control* - MVCC) otimista [Ramakrishnan and Gehrke 2002, Hazenberg and Hemminga 2011].

Para executar uma transação serialmente, o banco é divido em partições. E quando uma transação for executada, ela deve adquirir acesso exclusivo a todas as partições que precisa. Nessa estratégia, as transações podem ser executadas em paralelo em partições diferentes com *cores* diferentes [Stonebraker and Weisberg 2013, Malviya et al. 2014]. No MVCC, uma transação faz suas alterações em uma nova versão do dado original. Apenas a transação pode visualizar a versão criada por ela. Ao realizar *commit*, a transação não sobrescreve o dado original, ela apenas marca a nova versão do dado como atual e a original como obsoleta. Nessa abordagem não há partições e as transações podem executar concorrentemente [Diaconu et al. 2013, Freedman et al. 2014].

4.3.4. Durabilidade e recuperação após falhas

A maioria dos bancos de dados relacionais baseados em disco utilizam alguma forma de recuperação após falhas baseada no protocolo ARIES. O protocolo ARIES utiliza o esquema *write-ahead logging* (WAL) que escreve para um arquivo de *log* informações sobre atualizações feitas em uma página antes da confirmação de sua atualização. Cada registro no *log* possui um número sequencial (LSN) que serve para determinar a ordem em que cada ação foi executada no banco. ARIES utiliza um *log* físico onde são armazenadas as imagens da página antes e depois de sua atualização. Assim, o *log* possui informação suficiente para desfazer (UNDO) ou refazer (REDO) uma atualização. Após uma falha, ARIES executa os seguintes passos: (1) analisa o *log* para determinar o que deve ser recuperado; (2) refaz as transações que não tiveram suas atualizações descarregadas para o disco até o momento da falha; e (3) desfaz as transações que não terminaram. Após esses passos, o banco volta a um estado consistente igual a antes da falha e pode voltar a funcionar normalmente. Além disso, periodicamente, *checkpoints* são realizados para identificar transações que já foram descarregas para o disco e, consequentemente, diminuir a quantidade de *log* a ser analisada na recuperação [Mohan and Levine 1992].

Os bancos de dados em memória também devem guardar informações de *log* no disco para prover recuperação após falhas. Como o I/O é um gargalho, o *log* deve ser otimizado para não interferir no processamento normal das transações no sistema. Para diminuir o volume de *log*, apenas informações de REDO são gravadas em *group commit*, ou seja, as informações são acumuladas para envio em lote (até o tamanho de uma página, por exemplo) em um único I/O. Os sistemas em memória adotam um *log* lógico onde são gravadas descrições em alto nível de como as atualizações são feitas, como a inserção de um registro em uma tabela, por exemplo. O *log* lógico grava menos informações do que o físico evitando *overheads* ao sistema [Garcia-Molina and Salem 1992, Kim et al. 2012].

Para reduzir ainda mais o tráfico do *log*, alguns sistemas adotam um *log* chamado de *Command logging* (ou *Transaction logging*). *Command loggin* armazena apenas o nome da *store procedure* de uma transação e seus parâmetros. Assim, todas as ações de uma transação podem ser gravadas em um único registro. Como desvantagem do *Command loggin*, todas as transações devem ser definidas em forma de *store procedure*, ou seja, precisam ser definidas previamente [Malviya et al. 2014, Wu et al. 2017].

Os *checkpoints*, nos bancos em memória, propagam assincronamente as ações dos registros contidas no *log* para um arquivo no disco [Diaconu et al. 2013]. Em vez de propagar registros de *log*, alguns sistemas produzem arquivos persistentes (chamados de *snapshots*) também de forma assíncrona, que são equivalentes a um estado do banco materializado, através de uma técnica chamada de *copy-on-update*. O *checkpoint* pode ser considerado como o *backup* mais recente do banco de dados. Após uma falha, o gerenciador de recuperação carrega o *checkpoint* mais recente para a memória e, em seguida, executa os registros de *log* gravados logo após o último *checkpoint*. Produzir *checkpoints* diminui o tempo de recuperação do banco de dados, visto que, carregar para a memória as informações físicas do arquivo de *checkpoint* é mais rápido do que executar novamente as informações lógicas do *log* [Diaconu et al. 2013, DeWitt et al. 1984, Funke et al. 2014].

4.3.5. Processamento de consultas e compilação

O processamento de consultas é a atividade responsável por extrair informações de um banco de dados da forma mais eficiente possível. A Figura 4.6 ilustra as etapas do processamento de consultas. Quando uma consulta é submetida ao sistema, a primeira ação é traduzir a consulta em alto nível (SQL, por exemplo) para uma representação interna do banco (álgebra relacional, por exemplo) melhor entendida pelo sistema. Ao gerar a forma interna da consulta, são verificados a sintaxe da consulta e se os nomes das relações, atributos e etc. pertencem ao banco. Em seguida, o otimizador escolhe um plano de execução de consulta (sequência de operações) que tenha menos custo para o banco de dados. Diferentes planos de execução podem ser gerados e espera-se que o otimizador escolha o que execute mais rápido a consulta [Silberschatz et al. 2015].

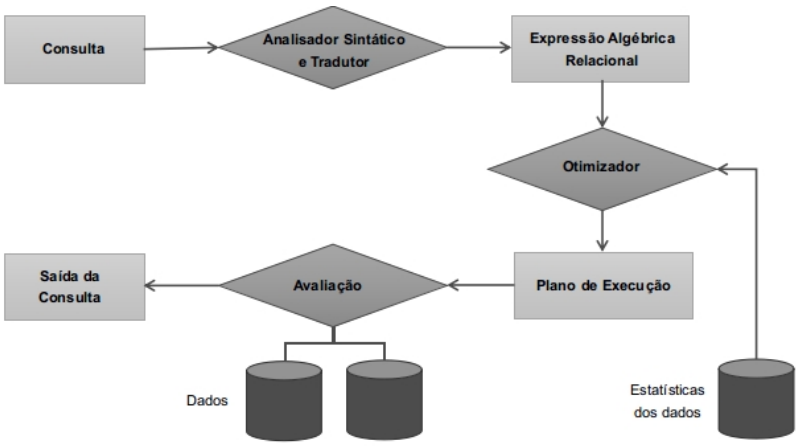


Figura 4.6. Etapas do processamento de consultas.
Fonte: [Silberschatz et al. 2015]

Os bancos de dados em disco e em memória possuem implementações semelhan-

tes nas etapas de análise e otimização, entretanto, diferem-se na execução da consulta. Os bancos em disco utilizam um modelo de iteração e interpretação em tempo de execução para executar as consultas. Essas técnicas servem para bancos de dados em discos, em que o I/O é o *overhead* dominante. Entretanto, elas são bastante custosas em bancos de dados em memória, em que ciclos de CPU a mais podem causar *overheads*. Os bancos em memória, em geral, compilam as consulta diretamente em código de máquina. Alguns bancos até restringem o uso de transações a *store procedures* para reduzir o processamento de consulta em tempo de execução [Faerber et al. 2017].

4.4. Bancos de dados em Memória

Com a tendência de residir os dados na memória, foram propostos vários tipos de bancos de dados em memória, desde os tradicionais bancos relacionais até os bancos NoSQL. Os bancos de dados relacionais utilizam o modelo relacional em que os itens de dados são organizados em tuplas de tabelas/relações com garantias ACID. São exemplos de Bancos de Dados em Memória Relacionais: SAP HANA [Färber et al. 2012], IBM DB2 with BLU Acceleration [Raman et al. 2013], HStore/VoltDB [Malviya et al. 2014], Oracle TimesTen [TimesTen Team 1999], P*Time [Cha and Song 2004], Microsoft Hekaton [Diaconu et al. 2013], ClustRa [Hvasshovd et al. 1995], solidDB [Lindström et al. 2013], NuoDB [Brynko 2012], eXtremeDB [McObject 2010], Pivotal SQLFire [Pivotal 2013], MemSQL [Memsql 2012], HyPer [Kemper and Neumann 2011], Silo [Tu et al. 2013], HYRISE [Grund et al. 2010], Dalí [Bohannon et al. 1997], DataBlitz [Baulier et al. 1999], Crescando [Unterbrunner et al. 2009], System K [Whitney et al. 1997] e MySQL Cluster NDB [Oracle 2004].

Os bancos de dados NoSQL geralmente estruturam seus dados como árvores, grafos ou chave/valor; e a linguagem de consulta, em geral, não é SQL. NoSQL é motivado pela sua simplicidade, escalabilidade horizontal, alta disponibilidade e tolerância a falhas; em detrimento da consistência. Alguns Sistemas de Bancos de Dados NoSQL são parcialmente em memória, pois usam arquivos de memória mapeada para armazenar dados de tal forma que o dado possa ser acessado como se estivesse na memória. Seguem alguns exemplos de bancos de dados NoSQL em memória: MongoDB [MongoDB 2009], MonetDB [Boncz et al. 2005], MDB [Chu 2011], MemepiC [Cai et al. 2014], RAMCloud [Ousterhout et al. 2010], Trinity [Shao et al. 2013], Redis [Sanfilippo and Noordhuis 2009] e Bitsy [Brown 2012].

Existe a categoria de Sistemas de *Cache* em Memória que são usados como uma *chace* entre o servidor de aplicação e o banco de dados adjacente. Esses sistemas são utilizados comumente por empresas como Facebook, Twitter e Wikipedia para prover bons serviços [Zhang et al. 2015]. Muitos sistema em *cache* tem sido desenvolvidos para vários propósitos. Por exemplo, existem sistemas em *cache* para uso geral, como Memcached [Fitzpatrick and Vorobey 2003] e BigTable Cache [Chang et al. 2008]; sistemas destinados a acelerar serviços de análise, como PACMan [Ananthanarayanan et al. 2012] e GridGain [Team 2007]; sistemas projetados para suportar *frameworks* específicos, como NCache [alachisoft 2005] para .NET e Velocity/AppFabric [Sampathkumar et al. 2009] para servidores Windows; sistemas que suportam semântica transacional rígida, como TxCache [Ports et al. 2010]; e *cache* de rede, como HashCache [Badam et al. 2009].

O advento de *Big Data* impulsionou o desenvolvimento de Sistemas de Processamento/Análise de Dados em Memória cujo objetivo principal é analisar grandes quantidades de dados em um pequeno intervalo de tempo [Zhang et al. 2015]. Existem basicamente dois tipos de sistemas de processamento de dados em memória: sistemas de análise de dados (por exemplo, Spark [Zaharia et al. 2012], Piccolo [Power and Li 2010], SINGA [Felber et al. 2008], Pregel [Malewicz et al. 2010], GraphLab [Malewicz et al. 2010], Mammoth [Shi et al. 2015], Phoenix [Yoo et al. 2009], GridGain [Team 2007]); e sistemas de processamento de dados em tempo real (por exemplo: Storm [BackType and Twitter 2011], Yahoo! S4 [Neumeyer et al. 2010], Spark Streaming [Zaharia et al. 2013], MapReduce Online [Condie et al. 2010]).

No restante dessa sessão serão abordados alguns sistemas de bancos de dados em memória relacionais, mas, antes disso, será dado um breve histórico do desenvolvimento de bancos de dados em memória.

4.4.1. Histórico

O desenvolvimento e pesquisa em bancos de dados em memória não é recente, começaram ainda no início da década de 80, como em IMS/Fast Path [Gawlick and Kinkade 1985, Strickland et al. 1982], MM-DBMS [Lehman and Carey 1986, Lehman and Carey 1987], MARS [Eich 1987, Gruenwald and Eich 1991], System M [Gruenwald and Eich 1991, Salem and Garcia-Molina 1990], TPK [Li and Naughton 2000], OBE [Bitton et al. 1987, Whang and Krishnamurthy 1990] e HALO [Garcia-Molina and Salem 1992, Eich 1987]. Muitas dessas pesquisas tentavam otimizar o uso da memória em bancos de dados em disco. Entretanto, a pouca capacidade de armazenamento e o custo elevando das memórias nessa época inviabilizaram a ampla adoção dos sistemas em memória.

A partir da década de 90, os avanços tecnológicos no *hardware* (principalmente na memória RAM e processamento *multicore*) e o seu barateamento impulsionaram o desenvolvimento e a pesquisa em sistemas de bancos de dados em memória. O bancos de dados desenvolvidos na década de 90 (por exemplo: ClustRa, P*Time, Dalí, DataBlitz, System K, TimesTen) já começaram a refletir as tendências e escolhas arquiteturas utilizadas nos bancos de dados em memória mais modernos, como: Hekaton, VoltDB, HyPer e SAP HANA [Garcia-Molina and Salem 1992, Eich 1987].

4.4.2. TimesTen

O TimesTen da Oracle é um banco de dados relacional em memória capaz de prover baixa latência e alto taxa de *throughput* requeridos pela indústria e empresas contemporâneas, como telecomunicações, mercado de capital e defesa, por exemplo. São exemplos de empresas que utilizam TimesTen: Ericsson, Dell, Bank of America Merrill Lynch, United States Postal Service, entre outras. Aplicativos podem acessar o TimesTen usando SQL padrão via *drivers* JDBC, ODBC e ODP.NET (*Oracle Data Provider for .NET*), além das *interfaces* Oracle PL/SQL, OCI (*Oracle Call Interface*), Pré-Compilador Pro*C/C++, e TTClasses (*TimesTen C++ Interface Classes*). A Figura 4.7 esquematiza a arquitetura de TimesTen. TimesTen pode ser instalado separadamente do banco de dados Oracle (chamado de TimesTen In-Memory Database ou simplesmente TimesTen) ou como uma *cache* para um bancos de dados Oracle (chamado de TimesTen Application-Tier Database

ou simplesmente TimesTen Cache). Implantado na camada de aplicação como um sistema embarcado, TimesTen opera em um banco de dados que cabe inteiramente na memória. TimesTen Cache é uma opção para melhora de desempenho de subconjuntos de uma banco de dados Oracle. Atualizações nas tabelas TimesTen Cache são sincronizados para tabelas Oracle automaticamente [TimesTen 2018]. No restante do texto vamos utilizar apenas TimesTen para nos referir tanto a TimesTen como a TimesTen Cache.

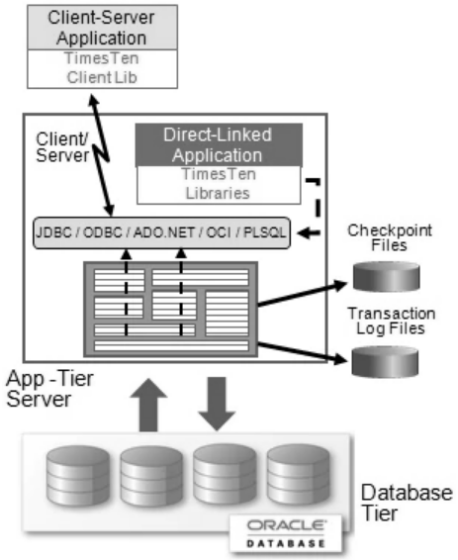


Figura 4.7. Arquitetura do banco de dados TimesTen.
Fonte: [TimesTen 2018]

TimesTen provê durabilidade e recuperação após falhas através de *log* de transações e *checkpoints*. Registros de *log* são escritos assincronamente ou sincronamente para o disco. Log assíncrono é aplicado em sistemas que exigem o máximo de *throughput* possível. Em casos onde a integridade dos dados deve ser preservada, o *log* síncrono deve ser utilizado. TimesTen executa periodicamente uma operação chamada de Fuzzy Checkpoint para gravar uma imagem do banco no disco. Essa operação é feita em *background* e causa poucos impactos a funcionamento normal do banco de dados. TimesTen também possui um *checkpoint* (chamado de Transaction-Consistent Checkpoint) que bloqueia exclusivamente o banco de dados para gravar uma imagem do banco para o disco. Esse *checkpoint* deve ser iniciado a partir de uma aplicação. Para recuperar o banco após uma falha, é necessário carregar o último *checkpoint* para memória e, após isso, executar os registros de *log* gravados após esse *checkpoint*. Adicionalmente, TimesTen pode replicar seus dados para prover alta-disponibilidade em caso de falhas [TimesTen 2018].

TimesTen possui um otimizador de consulta baseado em custos que tenta escolher o melhor plano de execução da consulta baseado em fatores como:

- presença de índices: *hash*, *bitmap* e *range*
- estatísticas sobre os dados: número de linhas e colunas de uma tabela; existência de chave primária em uma tabela; e tamanho e configuração de índices existentes.

- métodos para percorrer tabelas: *full table scan*, *rowid lookup*, *range index scan*, *bitmap index lookup*, e *hash index lookup*.
- algoritmos de junção: *nested loop joins*, *nested loop joins with indexes*, e *merge join*.

Além de tentar escolher a melhor estratégia para executar uma consulta com base nos fatores existentes, o otimizador também pode criar índices temporariamente. O otimizador também aceita *hints* que permitem decidir quais fatores podem influenciar na escolha do plano de execução da consulta [TimesTen 2018].

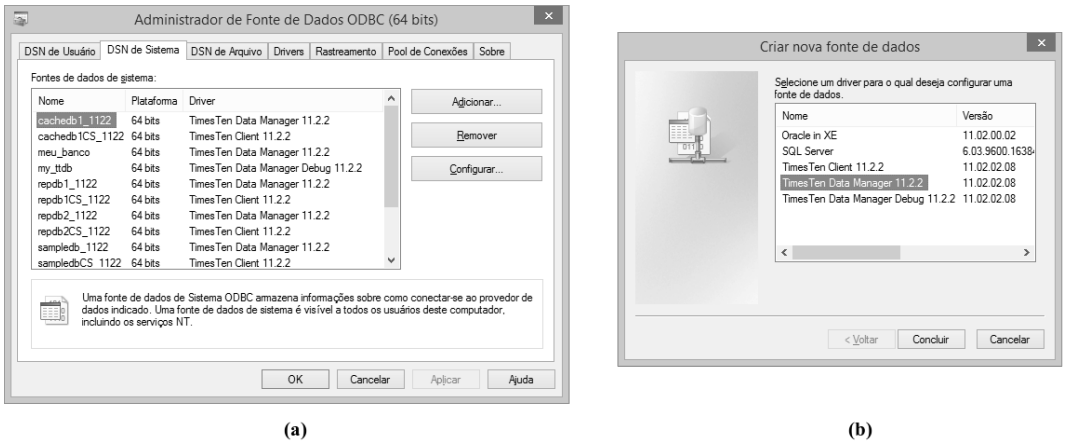
TimesTen dá suporte a bancos de dados compartilhados e coordena o acesso concorrente aos dados através de isolamento de transações e bloqueios. Esse banco provê opções para o usuário escolher um balanceamento entre tempo de resposta, *throughput* e semântica da transação. O isolamento dá a transação a impressão de estar executando sozinha, apesar de estar executando concorrentemente com outras transações no banco de dados. TimesTen suporta dois níveis de isolamento de transação: *read committed* e *serializable*. O isolamento *read committed* não utiliza bloqueios e é o modo de isolamento padrão. Nesse isolamento, as leituras são feitas em uma versão diferente dos dados das escritas, assim, não é necessário realizar bloqueios. No isolamento *serializable*, uma transação adquire bloqueios para ler/alterar dados e, consequentemente, outra transação devem esperar o desbloqueio desses dados para lê-los e/ou alterá-los. Os bloqueios realizados podem ser compartilhado, que permite leituras simultâneas de várias transações ao mesmo dado, mas não permite escritas; ou exclusivo, que permite a apenas uma transação alterar o dado durante o bloqueio. As aplicações podem selecionar o nível do bloqueio que desejam fazer: linha, tabela ou banco de dados. *Read committed* é útil para aplicações cujas transações executam longas varreduras de dados que podem conflitar com outras operações que precisam acessar uma linha percorrida. *Serializable* é útil para transações que requerem um nível mais forte de isolamento. [TimesTen 2018].

TimesTen tem distribuições para Windows e Linus e pode ser facilmente instalado, após ser baixado no site da Oracle através do link abaixo:
http://www.oracle.com/technetwork/database/database-technologies/timesten/downloads/.
Nas seções a seguir será dado um breve guia de como operar TimesTen.

4.4.2.1. Criar e manipular banco de dados TimesTen

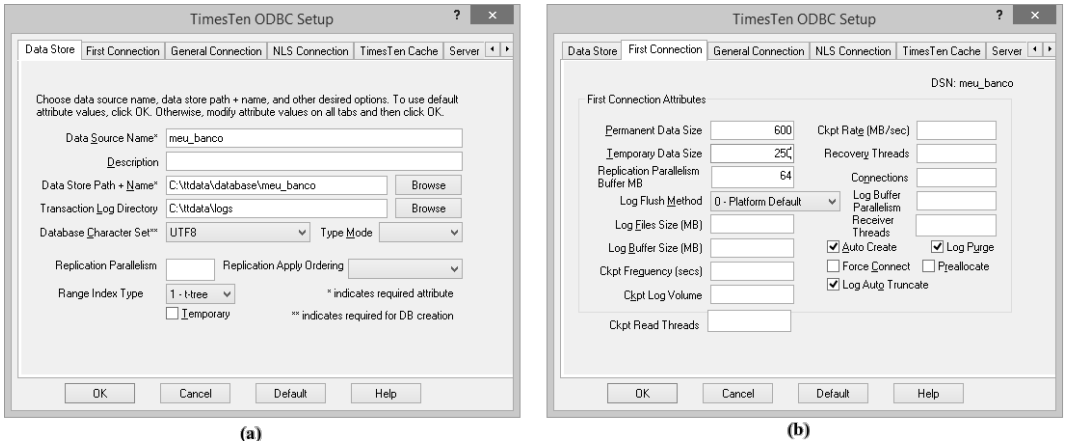
Bancos de dados TimesTen são acessados através de DSN (*Data Source Name*), que é um nome lógico para identificar um banco de dados. Cada DSN contem atributos que especificam as propriedades do banco de dados a ser criado ou acessado, tais como: nome, *path*, tamanho e etc.. Nessa seção será mostrado como configurar DSN na plataforma Windows. Um DSN pode ser criado e mantido através do "Administrador de Fonte de Dados ODBC"(Figura 4.8 (a)). Para criar uma nova fonte de dados deve ser clicado no botão "Adicionar..."da aba "DNS do Sistema"do "Administrador de Fonte de Dados ODBC". Na tela "Criar nova fonte de dados"(Figura 4.8 (b)) deve ser selecionado "TimesTen Data Manager 11.2.2"e, em seguida, clicado no botão "Concluir". Na aba "Data Source"da janela de configuração ODBC do TimesTen (Figura 4.9 (a)), devem ser setadas

as informações sobre a fonte de dados do banco: o nome do banco de dados em "Data Souce Name"; o diretório e o nome do banco de dados em "Data Source Path + Name"; o diretório dos arquivos de *log* do banco de dados em "Transaction Log Directory"; e o padrão de caracteres em "Database Character Set". No exemplo da Figura 4.9 (a) o nome do banco de dados é *meu_banco*; o *path* do banco é *C://ttdata/database/meu_banco*; o diretório de *logs* é *C://ttdata/database/logs*; e o padrão de caracteres segue o UTF-8. Na aba "First Connection"(Figura 4.9 (b)) devem ser setados, em MB, o tamanho máximo da base de dados em "Permanent Data Size"; e o tamanho máximo reservado a dados temporário em "Temporary Data Size". No exemplo da Figura 4.9 (b) o espaço reservado ao banco foi 600MB e o aos dados temporários foi 250MB. Após todos esses dados serem digitados, deve ser clicado no botão "Ok"para finalizar a configuração ODBC do banco.



(a) (b)

Figura 4.8. (a) Administrador de Fonte de Dados ODBC do Windows; e (b) Criar nova fonte de dados.



(a) (b)

Figura 4.9. Configuração ODBC de um banco de dados TimesTen.

Um banco de dados TimesTen é criando em sua primeira conexão. A primeira conexão a um banco já existente carrega-o na memória e, dependendo do tamanho da base de dados, esse carregamento pode demorar. Como exemplo, para criar e manipular um banco TimesTen é utilizada a ferramenta ttIsql, que é instalada juntamente com TimesTen

e pode ser acessada no Prompt de Comando através do comando *ttisql*. Outras ferramentas, como o SQLDeveloper, também podem ser utilizadas. No ttIsql, o comando *connect "dsn=<nome_do_banco>"* deve ser digitado para se conectar a um banco de dados. A Figura 4.10 mostra um exemplo de como se conectar ao banco chamado de *meu_banco* utilizando a ferramenta ttIsql. TimesTen utiliza o SQL padrão e , consequentemente, utiliza as instruções SQL habituais para manipular bancos de dados. A Figura 4.11 mostra instruções SQL para criar uma tabela e inserir e selecionar registros.

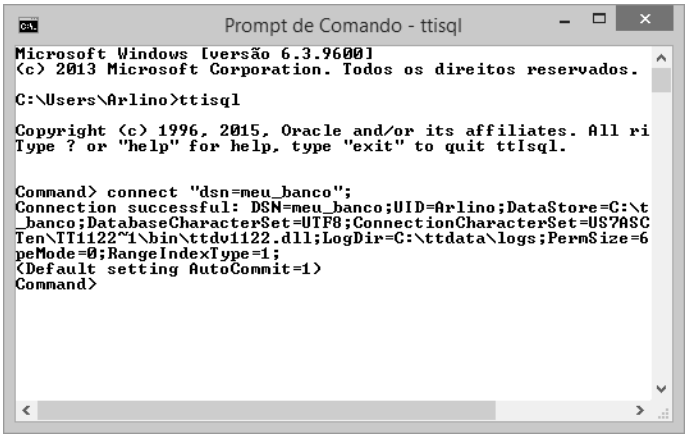


Figura 4.10. Conexão a um banco de dados TimesTen através da ferramenta ttIsql.

```
CREATE TABLE CUSTOMER (
   C_CUSTKEY NUMBER primary key,
   C_NAME VARCHAR2(25 BYTE),
   C_ADDRESS VARCHAR2(40 BYTE),
   C_PHONE VARCHAR2(15 BYTE) );

INSERT INTO CUSTOMER VALUES (1, 'Cliente', 'Rua ...', '9999-9999');

SELECT * FROM CUSTOMER;
```

Figura 4.11. Instruções SQL para criar uma tabela e inserir e selecionar registros.

4.4.3. Hekaton

Hekaton é uma *engine* do banco de dados Microsoft SQL Server que trabalho com cargas de trabalho OLTP residentes na memória. O nome oficial de Hekaton é *OLTP in-memory*. Uma tabela no Haketon (também chamada de tabela otimizada na memória) é armazenada inteiramente na memória e é durável. Um banco de dados pode conter tabelas Haketon (na memória) e tabelas SQL Server (no disco). Uma transação é acessada usando T-SQL (*Transact-SQL*) e pode ler/escrever em ambas, tabelas em memória e tabelas em disco. *Store procedures* que referenciam apenas tabelas Haketon podem ser compilados em código nativo de máquina. Os dados em Hekaton são livres de bloqueios e são implementados em multi-versionamento. Os índices também não utilizam bloqueios e podem ser

índices *hash*, *Bw-Tree* e *columnstore*. Haketon utiliza controle de concorrência MVCC otimista. O protocolo de controle de concorrência otimista não se preocupa em prevenir conflitos entre transações, ao invés disso, detecta conflitos em um processo de validação das atualizações de uma transação antes do *commit*. Se a validação falhar, a transação é abortada [Diaconu et al. 2013, Freedman et al. 2014, Larson et al. 2013].

Para atingir durabilidade e recuperação após falhas, Haketon utiliza *logs* e *checkpoints*. Apenas *logs* de REDO são produzidos, assim, quando uma transação é efetivada, informações sobre versões criadas e deletadas são enviados para o *log* em um *group commit*. Periodicamente, *checkpoints* são realizados durante a execução normal das transações no sistema. O *checkpoint* é armazenado em dois tipos de arquivos: (1) arquivos *data* que contem versões inseridas (geradas por *inserts* e *updates*); e (2) arquivos *delta* que armazenam informações sobre quais versões contidas em um arquivo *data* foram deletadas. Após uma falha, a recuperação de Haketon procura o arquivo de inventário do *checkpoint* que contem referências sobre todos os arquivos *data* e *delta* e, em seguida, carrega as versões contidas nos arquivos *data* utilizando os arquivos *delta* como filtro para versões deletadas. O par de arquivos *data*/*delta* é considerado a unidade de recuperação e permite uma recuperação altamente paralelizada. Após todos os arquivos *data* serem carregados na memória, os registros de *log* são executados a partir do *timestamp* do último *checkpoint* realizado [Diaconu et al. 2013, Faerber et al. 2017].

Haketon está disponível nas plataformas Windows 8 ou superiores e é instalado juntamente com o SQL Server a partir da versão 2014. Há uma versão *trial* do SQL Server disponível para *download* e instalação no *link*:
https://www.microsoft.com/pt-br/sql-server/sql-server-downloads
Nas seções a seguir será dado um breve guia de como operar Haketon.

4.4.3.1. Criar e manipular banco de dados Hekaton

O primeiro passo para utilizar tabelas Haketon é criar grupos de arquivos otimizados para memória. Esses arquivos retem um ou mais contêineres que contem arquivos *data* e/ou *delta*. Os grupos de arquivos podem ser criados durante a criação de um banco de dados ou posteriormente. Como exemplo, para criar e manipular um banco SQL Server Haketon será utilizada a ferramenta SQLCMD (Figura 4.12), que é instalada junta o SQL Server, e pode ser acessada através do Prompt de Comando. Digite o comando "*SQLCMD -E -S <endereço_do_servidor>\<nome_da_instância>*"no Prompt de Comando para conectar-se ao SQL Server [Haketon 2018]. No exemplo da Figura 4.12 é feita uma conexão ao servidor *localhost* na instância *MSSQLSERVER01*.

Depois de conectar-se ao SQL Server, na Figura 4.12, é criado um banco de dados chamado de *TPCH* com suporte a tabelas em memória utilizando o comando da Figura 4.13; em seguida, o banco *TPCH* é acessao através do comando "*use TPCH*"; e, finalmente, é criada uma tabela em memória chamada de *CUSTOMER* utilizando o comando da Figura 4.14. No comando da Figura 4.13 é criado o banco de dados *TPCH* com o arquivo *TPCH_data*, para armazenar os dados; e com o grupo de arquivos *TPCH_mod*, para os contêineres de otimização de memória. Os arquivos são armazenados no diretório *C:\HKData\TPCH*. No comando da Figura 4.14 é criada *CUSTOMER* que será uma ta-

bela em memória (através da expressão *MEMORY_OPTIMIZED = ON*) com durabilidade em disco (através da expressão *DURABILITY = SCHEMA_AND_DATA*). Os comandos para inserir, alterar e excluir registros de uma tabela utilizam o SQL padrão. Adicionalmente, SQLCMD exige que cada comando seja finalizado e executado através da palavra *GO* [Haketon 2018].

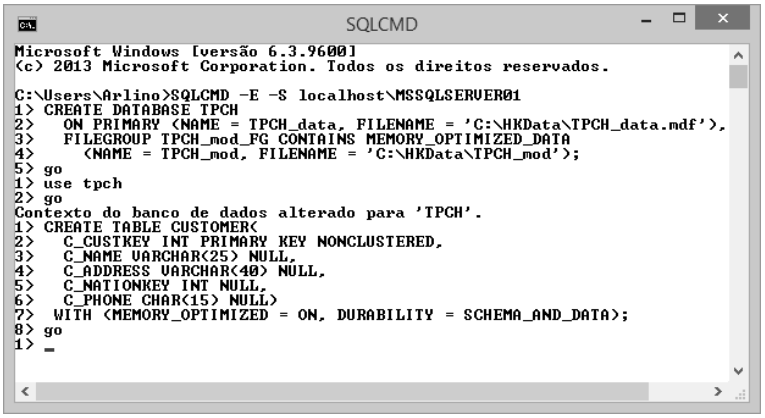


Figura 4.12. Conexão, criação de banco de dados, e criação de tabela em memória no SQL Server Haketon através da ferramenta SQLCMD.

```
CREATE DATABASE TPCH
  ON PRIMARY (NAME = TPCH_data, FILENAME = 'C:\HKData\TPCH_data.mdf'),
  FILEGROUP TPCH_mod_FG CONTAINS MEMORY_OPTIMIZED_DATA
  (NAME = TPCH_mod, FILENAME = 'C:\HKData\TPCH_mod');
```

Figura 4.13. Instrução SQL para criar um banco de dado com suporte a tabelas em memória no SQL Server Haketon.

```
CREATE TABLE CUSTOMER
  C_CUSTKEY INT PRIMARY KEY NONCLUSTERED,
  C_NAME VARCHAR(25) NULL,
  C_ADDRESS VARCHAR(40) NULL,
  C_NATIONKEY INT NULL,
  C_PHONE CHAR(15) NULL)
  WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Figura 4.14. Instrução SQL para criar uma tabela durável em memória no SQL Server Haketon.

4.4.4. VoltDB

VoltDB é um banco de dados em memória projetado para cargas de trabalho OLTP que pode ser implantado em um *cluster* e cujo *design* é baseado no H-Store [Kallman et al. 2008] (versão acadêmica). VoltDB particiona seus dados (Figura 4.15 (a)) e, consequentemente,

pode suportar bases de dados maiores do que a memória disponível em um único nó. Além disso, os nós podem ser replicados para implementar alta disponibilidade e tolerância a falhas. As transações são executadas serialmente (Figura 4.15 (b)), ou seja, uma transação deve ter acesso exclusivo a todas as partições que precisa antes de começar a executar. Cada transação deve ser um *store procedure* pré definida [Malviya et al. 2014, Faerber et al. 2017, Stonebraker and Weisberg 2013].

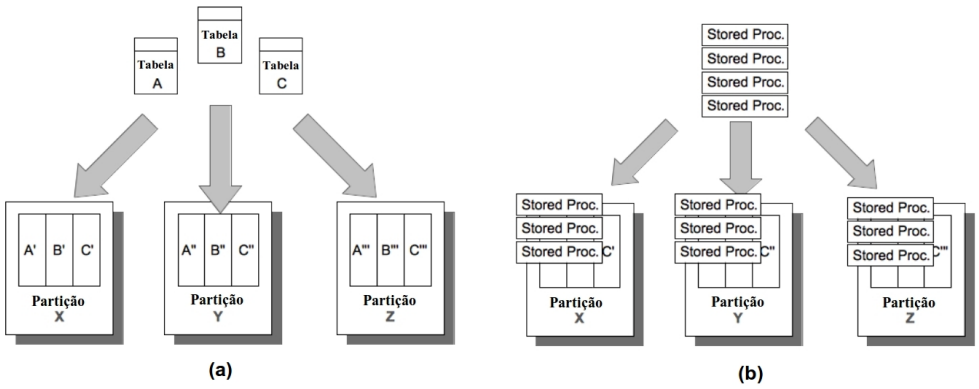


Figura 4.15. (a) particionamento de dados e (b) execução serial no VoltDB.

VoltDB utiliza *command loggin*, descrito na Seção 4.3.4, e um *checkpoint* sem bloqueios consistente a nível de transação para prover tolerância a falhas. Uma transação executada em uma única partição grava suas ações em seu arquivo de *log*. Em uma transação distribuída (executadas em várias partições), apenas o nó que coordenada a execução da transação grava os registros de *log*. Os *logs* são escritos também para as replicas de um nó, se elas existirem. Checkpoints são executados periodicamente durante o funcionamento normal das transações no sistema. O *checkpoint* produz uma imagem do banco de dados para o disco chamada de *snapshot*. Após uma falha, o processo de recuperação carrega para a memória o último *snapshot* e, em seguida, os registros de *log* gravados depois do *snapshot* são executados [Malviya et al. 2014, Faerber et al. 2017].

VoltDB está disponível nas plataforma Linux. Há uma versão *trial* do VoltDB disponível para *download* no *link*:
https://www.voltdb.com/try-voltdb/
Na seção a seguir será dado um breve guia de como operar VoltDB em um único nó.

4.4.4.1. Criar e manipular banco de dados VoltDB

A instalação de VoltDB é simples e requer apenas descompactar o arquivo de instalação. A descompactação pode ser feita utilizando as ferramentas gráficas de compactação/descompactação de arquivos disponíveis no Linux ou através de linha de comando em Terminal, como o *bash*, por exmeplo. Por exemplo, o comando "*tar -zxvf voltdb-ent-8.0.tar.gz -C $HOME/*"faz a descompactação do arquivo *voltdb-ent-8.0.tar.gz* para o diretório pessoal do usuário. É recomendado que o nome do diretório descompactado do VoltDB seja mudado para um nome mais simples como simplesmente *voltdb* [VoltDB 2018]. A Figura 4.16 sumariza a sequência de comandos a serem digitados em

um Terminal para instalação do VoltDB.

```
$ tar -zxvf voltdb-ent-8.0.tar.gz -C $HOME/
$ sudo mv voltdb-ent-8.0 voltdb
```

Figura 4.16. Passos para instalação do VoltDB no Linux via Terminal.

VoltDB possui *scripts* que facilitam o processo de desenvolvimento e implantação de aplicações. Esses *scripts* estão na pasta */bin* que está no diretório onde o VoltDB foi instalado. Para tornar esses *scripts* disponíveis em forma de comandos no Terminal do Linux é necessário adicionar o *path* do diretório */bin* às variáveis de ambiente do sistema utilizando, por exemplo, o comando "*export PATH="$PATH:$HOME/voltdb/bin*"no Terminal. A Figura 4.17 ilustra o comando para adicionar o *path* "*$HOME/voltdb/bin*"às variáveis de ambiente. Para não precisar adicionar o *path* a cada vez que uma sessão for criada, o comando da Figura 4.17 pode ser adicionado no final do arquivo "*$HOME/.bashrc*".

```
$ export PATH="$PATH:$HOME/voltdb/bin
```

Figura 4.17. Comando para adicionar o *path* do diretório */bin* do VoltDB às variáveis de ambiente no Linux via Terminal.

Antes de iniciar o VoltDB, o diretório *root* do sistema deve ser inicializado. Essa diretório armazena informações de configuração de dados, *logs* e outras informações relacionadas ao disco. Após o diretório *root* ser inicializado, o VoltDB pode ser iniciado. Os comandos "*voltdb init*"e "*voltdb start*"devem ser digitados no Terminal para inicializar o diretório *root* e inciar o VoltDB, respectivamente. Adicionalmente, para parar o VoltDB, o comando "*voltadmin shutdown*"deve ser utilizado. A Figura 4.18 mostra a execução dos comandos para inicializar o diretório *root* e para iniciar o VoltDB em um Terminal mensagens retornadas por esses comandos.

```
$ voltdb init
$ voltdb start
$ voltadmin shutdown
```

Figura 4.18. Comandos para inicializar o diretório *root* do VoltDB e para iniciá-lo e pará-lo.

Após iniciado o VoltDB, podem ser executados comandos SQL, como: criação de tabelas e inserção, alteração e remoção de linhas de tabelas. VoltDB utiliza o SQL padrão, como exemplificado na Figura 4.11. A ferramenta SQLCMD pode ser utilizada para executar comandos SQL no VoltDB e pode ser acessada através do comando *sqlcmd* no

Terminal. Adicionalmente, VoltDB possui a ferramenta de gerenciamento VoltDB Management Center que pode ser acessada digitando a *url http://<nome_do_servidor>:8080* no *browser*, onde *<nome_do_servidor>* é o nome do servidor Web do VoltDB, como *localhost*, por exemplo.

4.5. Conclusões

Muitas aplicações contemporâneas tem requerido serviços de latência muito baixa e com possibilidades de análise em tempo real. Os sistemas convencionais baseados não tem conseguido atender aos requerimentos das aplicações contemporâneas. Assim, sistemas em memória tem sido utilizados como alternativa para atender aos requerimentos dessas aplicações, visto que, os bancos de dados em memória armazenam seus dados diretamente na memória proporcionando altas velocidades de acesso. Entretanto, devido a grande diferença entre o disco e a memória, todos os aspectos de gerenciamento em banco de dados em memória devem ser repensado. Este trabalho discutiu os principais conceitos e técnicas que diferenciam os bancos residentes em memória dos bancos residentes em disco, auxiliando na tomada de decisão de tipo de banco é mais adequado a um determinado sistema.

Referências

[alachisoft 2005] alachisoft (2005). Ncache: In-memory distributed cache for .net. *http://www.alachisoft.com/ncache/*. Acessado em: 14/04/2018.

[Ananthanarayanan et al. 2012] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. (2012). Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association.

[Arulraj et al. 2015] Arulraj, J., Pavlo, A., and Dulloor, S. R. (2015). Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM.

[BackType and Twitter 2011] BackType and Twitter (2011). Storm. *https://storm.incubator.apache.org*. Acessado em: 18/04/2018.

[Badam et al. 2009] Badam, A., Park, K., Pai, V. S., and Peterson, L. L. (2009). Hashcache: Cache storage for the next billion. In *NSDI*, volume 9, pages 123–136.

[Baulier et al. 1999] Baulier, J., Bohannon, P., Gogate, S., Gupta, C., and Haldar, S. (1999). Datablitz storage manager: main-memory database performance for critical applications. In *ACM SIGMOD Record*, volume 28, pages 519–520. ACM.

[Bitton et al. 1987] Bitton, D., Hanrahan, M. B., and Turbyfill, C. (1987). Performance of complex queries in main memory database systems. In *Data Engineering, 1987 IEEE Third International Conference on*, pages 72–81. IEEE.

[Bohannon et al. 1997] Bohannon, P., Lieuwen, D., Rastogi, R., Silberschatz, A., Seshadri, S., and Sudarshan, S. (1997). The architecture of the dali main-memory storage manager. In *Multimedia Database Management Systems*, pages 23–59. Springer.

[Boncz et al. 2005] Boncz, P. A., Zukowski, M., and Nes, N. (2005). Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237.

[Brown 2012] Brown, M. C. (2012). *Getting Started with Couchbase Server: Extreme Scalability at Your Fingertips*. "O'Reilly Media, Inc.".

[Brynko 2012] Brynko, B. (2012). Nuodb: Reinventing the database. *Information Today*, 29(9):9–9.

[Cai et al. 2014] Cai, Q., Zhang, H., Chen, G., Ooi, B. C., and Tan, K.-L. (2014). Memepic: Towards a database system architecture without system calls.

[Cascaval et al. 2008] Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. (2008). Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40.

[Cha and Song 2004] Cha, S. K. and Song, C. (2004). P* time: Highly scalable oltp dbms for managing update-intensive stream workload. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1033–1044. VLDB Endowment.

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

[Chatzistergiou et al. 2015] Chatzistergiou, A., Cintra, M., and Viglas, S. D. (2015). Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508.

[Chen et al. 2001] Chen, S., Gibbons, P. B., and Mowry, T. C. (2001). *Improving index performance through prefetching*, volume 30. ACM.

[Chu 2011] Chu, H. (2011). Mdb: A memory-mapped database and backend for openldap. In *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, page 35.

[Condie et al. 2010] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. (2010). Mapreduce online. In *Nsdi*, volume 10, page 20.

[DeWitt et al. 1984] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. (1984). *Implementation techniques for main memory database systems*, volume 14. ACM.

[Diaconu et al. 2013] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. (2013). Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM.

[Driskill-Smith 2010] Driskill-Smith, A. (2010). Latest advances and future prospects of stt-ram. In *Non-Volatile Memories Workshop*, pages 11–13.

[Dulloor et al. 2014] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. (2014). System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM.

[Eich 1987] Eich, M. H. (1987). A classification and comparison of main memory database recovery techniques. In *Data Engineering, 1987 IEEE Third International Conference on*, pages 332–339. IEEE.

[Elmasri and Navathe 2010] Elmasri, R. and Navathe, S. (2010). *Fundamentals of database systems*. Addison-Wesley Publishing Company.

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

[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 Data Eng. Bull.*, 35(1):28–33.

[Felber et al. 2008] Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM.

[Fitzpatrick and Vorobey 2003] Fitzpatrick, B. and Vorobey, A. (2003). Memcached: A distributed memory object caching system. *http://memcached.org/*. Acessado em: 14/04/2018.

[Freedman et al. 2014] Freedman, C., Ismert, E., and Larson, P.-Å. (2014). Compilation in the microsoft sql server hekaton engine. *IEEE Data Eng. Bull.*, 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 engineering*, 4(6):509–516.

[Gawlick and Kinkade 1985] Gawlick, D. and Kinkade, D. (1985). Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8(2):3–10.

[Gruenwald and Eich 1991] Gruenwald, L. and Eich, M. H. (1991). *MMDB reload algorithms*, volume 20. ACM.

[Gruenwald and Eich 1994] Gruenwald, L. and Eich, M. H. (1994). Mmdb reload concerns. *Information sciences*, 76(1):151–176.

[Grund et al. 2010] Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., and Madden, S. (2010). Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116.

[Haketon 2018] Haketon (2018). Otimizar o desempenho usando tecnologias in-memory no banco de dados sql. *https://docs.microsoft.com/pt-br/azure/sql-database/sql-database-in-memory*. Acessado em: 30/04/2018.

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

[Hvasshovd et al. 1995] Hvasshovd, S.-O., Torbjørnsen, Ø., Bratsberg, S. E., and Holager, P. (1995). The clustra telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477. Morgan Kaufmann Publishers Inc.

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

[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 *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE.

[Kim et al. 2012] Kim, J.-J., Kang, J.-J., and Lee, K.-Y. (2012). Recovery methods in main memory dbms. *International journal of advanced smart convergence*, 1(2):26–29.

[Knight 1986] Knight, T. (1986). An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112. ACM.

[Larson et al. 2013] Larson, P.-Å., Zwilling, M., and Farlee, K. (2013). The hekaton memory-optimized oltp engine. *IEEE Data Eng. Bull.*, 36(2):34–40.

[Lehman and Carey 1986] Lehman, T. J. and Carey, M. J. (1986). *Query processing in main memory database management systems*, volume 15. ACM.

[Lehman and Carey 1987] Lehman, T. J. and Carey, M. J. (1987). *A recovery algorithm for a high-performance memory-resident database system*, volume 16. ACM.

[Leis et al. 2014a] Leis, V., Boncz, P., Kemper, A., and Neumann, T. (2014a). Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754. ACM.

[Leis et al. 2014b] Leis, V., Kemper, A., and Neumann, T. (2014b). Exploiting hardware transactional memory in main-memory databases. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 580–591. IEEE.

[Li and Naughton 2000] Li, K. and Naughton, J. F. (2000). Multiprocessor main memory transaction processing. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 177–187. IEEE Computer Society Press.

[Li et al. 2013] Li, Y., Pandis, I., Mueller, R., Raman, V., and Lohman, G. M. (2013). Numa-aware algorithms: the case of data shuffling. In *CIDR*.

[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 Data Eng. Bull.*, 36(2):14–20.

[Lintzmayer et al.] Lintzmayer, C. N., Theodoro, E., and Sambinelli, M. Memória transacional.

[Maas et al. 2013] Maas, L. M., Kissinger, T., Habich, D., and Lehner, W. (2013). Buzzard: A numa-aware in-memory indexing system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1285–1286. ACM.

[Malewicz et al. 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM.

[Malviya et al. 2014] Malviya, N., Weisberg, A., Madden, S., and Stonebraker, M. (2014). Rethinking main memory oltp recovery. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 604–615. IEEE.

[Mao et al. 2012] Mao, Y., Kohler, E., and Morris, R. T. (2012). Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM.

[McObject 2010] McObject (2010). extremedb database system. *http://www.mcobject.com/extremedbfamily.shtml*. Acessado: 14/04/2018.

[Memsql 2012] Memsql (2012). Memsql inc. *http://www.memsql.com/*. Acessado em: 14/04/2018.

[Mohan and Levine 1992] Mohan, C. and Levine, F. (1992). *ARIES/IM: an efficient and high concurrency index management method using write-ahead logging*, volume 21. ACM.

[MongoDB 2009] MongoDB (2009). Mongodb. *http://www.mongodb.org/*. Acessado em: 14/04/2018.

[Neumeyer et al. 2010] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE.

[Oracle 2004] Oracle (2004). Mysql cluster ndb. *http://www.mysql.com/*. Acessado em: 14/04/2018.

[Oukid et al. 2014] Oukid, I., Booss, D., Lehner, W., Bumbulis, P., and Willhalm, T. (2014). Sofort: A hybrid scm-dram storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 8. ACM.

[Ousterhout et al. 2010] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., et al. (2010). The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 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.

[Pivotal 2013] Pivotal (2013). Pivotal sqlfire. *http://www.vmware.com/products/vfabric-sqlfire/overview.html*. Acessado em: 14/04/2018.

[Porobic et al. 2014] Porobic, D., Liarou, E., Tozun, P., and Ailamaki, A. (2014). Atrapos: Adaptive transaction processing on hardware islands. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 688–699. IEEE.

[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.

[Ports et al. 2010] Ports, D. R., Clements, A. T., Zhang, I., Madden, S., and Liskov, B. (2010). Transactional consistency and automatic management in an application data cache.

[Power and Li 2010] Power, R. and Li, J. (2010). Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14.

[Ramakrishnan and Gehrke 2002] Ramakrishnan, R. and Gehrke, J. (2002). *Database management systems*. McGraw Hill.

[Raman et al. 2013] Raman, V., Attaluri, G., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G. M., et al. (2013). Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091.

[Rao and Ross 2000] Rao, J. and Ross, K. A. (2000). Making b+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM.

[Raoux et al. 2008] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., et al. (2008). Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479.

[Robbins 2008] Robbins, S. (2008). Ram is the new disk... *https://www.infoq.com/news/2008/06/ram-is-disk/*. Acessado em: 14/04/2018.

[Saha et al. 2006] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. (2006). Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM.

[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 Engineering*, 2(1):161–172.

[Sampathkumar et al. 2009] Sampathkumar, N., Krishnaprasad, M., and Nori, A. (2009). Introduction to caching with windows server appfabric. *Microsoft Corporation, Albuquerque, NM, USA, Tech. Rep*.

[Sanfilippo and Noordhuis 2009] Sanfilippo, S. and Noordhuis, P. (2009). Redis. *http://redis.io*. Acessado em: 14/04/2018.

[Shao et al. 2013] Shao, B., Wang, H., and Li, Y. (2013). Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM.

[Shavit and Touitou 1997] Shavit, N. and Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10(2):99–116.

[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 Transactions on Parallel and Distributed Systems*, 26(8):2300–2315.

[Silberschatz et al. 2015] Silberschatz, A., Korth, H. F., Sudarshan, S., et al. (2015). *Database system concepts*, volume 4. McGraw-Hill New York.

[Stonebraker and Weisberg 2013] Stonebraker, M. and Weisberg, A. (2013). The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 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(4):490–510.

[Strukov et al. 2008] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. (2008). The missing memristor found. *nature*, 453(7191):80.

[Team 2007] Team, G. (2007). Gridgain: In-memory computing platform. *http://gridgain.com/*. Acessado em: 14/04/2018.

[TimesTen 2018] TimesTen (2018). Oracle timesten in-memory database documentation. *https://docs.oracle.com/cd/E21901_01/index.html*. Acessado em: 24/04/2018.

[TimesTen Team 1999] TimesTen Team, C. (1999). In-memory data management for consumer transactions the timesten approach. In *ACM SIGMOD Record*, volume 28, pages 528–529. 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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM.

[Unterbrunner et al. 2009] Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., and Kossmann, D. (2009). Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717.

[VoltDB 2018] VoltDB (2018). Voltdb documentation - using voltdb. *https://docs.voltdb.com/UsingVoltDB/*. Acessado em: 04/05/2018.

[Whang and Krishnamurthy 1990] Whang, K.-Y. and Krishnamurthy, R. (1990). Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems (TODS)*, 15(1):67–95.

[Whitney et al. 1997] Whitney, A., Shasha, D., and Apter, S. (1997). High volume transaction processing without concurrency control, two phase commit, sql or c++.

[Wu et al. 2017] Wu, Y., Guo, W., Chan, C.-Y., and Tan, K.-L. (2017). Fast failure recovery for main-memory dbmss on multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 267–281. 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE.

[Zaharia et al. 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.

[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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

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

[Zilles and Rajwar 2007] Zilles, C. and Rajwar, R. (2007). Implications of false conflict rate trends for robust software transactional memory. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 15–24. IEEE.

Sobre os autores

Arlino Henrique Magalhães de Araújo

Possui mestrado em Ciência da Computação pela Universidade Federal do Ceará - UFC (2013) e graduação em Bacharelado em Ciência da Computação pela Universidade Federal do Piauí - UFPI (2004). Atualmente, trabalha como professor no curso de Sistemas de Informação no Centro de Educação Aberta à Distância da UFPI e está cursando doutorado na UFC. Tem como áreas de interesse Banco de Dados e Engenharia de Software atuando principalmente nos seguintes temas: sintonia automática de bancos de dados e bancos de dados em memória. Detalhes em: http://lattes.cnpq.br/6618696720418338.

José Maria da Silva Monteiro Filho

Possui graduação em Bacharelado em Computação pela Universidade Federal do Ceará - UFC (1998), mestrado em Ciência da Computação pela UFC (2001) e doutorado em Informática pela Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio (2008). Atualmente é professor na UFC. Tem experiência na área de Ciência da Computação, com ênfase em Banco de Dados e Engenharia de Software, atuando principalmente nos seguintes temas: sintonia automática de bancos de dados, bancos de dados em nuvem, dados ligados na Web e qualidade de software. Detalhes em: http://lattes.cnpq.br/9790693300026949.

Ângelo Roncalli Alencar Brayner

Ângelo Brayner concluiu sua graduação em Ciência da Computação em 1988 pela Universidade Federal do Ceará - UFC. Em 1994, obteve o título de Mestre em Ciência da Computação pela Universidade Estadual de Campinas. Em 1999, concluiu o doutorado em Ciência da Computação pela Universität Kaiserslautern, Alemanha, sob a orientação do Prof. Dr.-Ing. Theo Härder. Atualmente é professor titular na UFC. Em 2014, ministrou uma disciplina na Universidade de Stuttgart (programa de mestrado - IMSE), Alemanha, como professor visitante. É autor do livro Transaction Management in Multidatabase Systems, publicado pela Shaker-Verlag, Alemanha, em 1999. Prof. Ângelo Brayner é autor de mais de 50 artigos publicados em periódicos e conferências internacionais e nacionais. Detalhes em: http://lattes.cnpq.br/3895469714548887.