**UNIVERSIDADE FEDERAL DO CEARÁ**

**CENTRO DE CIÊNCIAS**

**DEPARTAMENTO DE COMPUTAÇÃO**

**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**MESTRADO EM CIÊNCIA DA COMPUTAÇÃO**

**GUSTAVO DE OLIVEIRA MORAES**

# EBRES: UMA ESTRATÉGIA INTELIGENTE DE SUBSTITUIÇÃO DE PÁGINAS PARA BANCO DE DADOS

**FORTALEZA**

**2022**

GUSTAVO DE OLIVEIRA MORAES

# EBRES: UMA ESTRATÉGIA INTELIGENTE DE SUBSTITUIÇÃO DE PÁGINAS PARA BANCO DE DADOS

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

Orientador: Prof. Angelo Roncalli Alencar Brayner, Dr.-Ing.

Coorientador: Prof. José de Aguiar Moraes Filho, Dr.-Ing.

FORTALEZA

2022

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

---

M82e Moraes, Gustavo de Oliveira.
EBRES : uma estratégia inteligente de substituição de páginas para banco de dados / Gustavo de Oliveira Moraes. – 2022.
111 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2022.
Orientação: Prof. Dr. Angelo Roncalli Alencar Brayner.
Coorientação: Prof. Dr. José de Aguiar Moraes Filho.

1. Bancos de dados. 2. Política de substituição de buffer. 3. Técnicas de predição. I. Título.
CDD 005

GUSTAVO DE OLIVEIRA MORAES

# EBRES: UMA ESTRATÉGIA INTELIGENTE DE SUBSTITUIÇÃO DE PÁGINAS PARA BANCO DE DADOS

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

Aprovada em: 28 de Outubro de 2022

BANCA EXAMINADORA

---

Prof. Angelo Roncalli Alencar Brayner,
Dr.-Ing. (Orientador)
Universidade Federal do Ceará (UFC)

---

Prof. José de Aguiar Moraes Filho,
Dr.-Ing. (Coorientador)
Serviço Federal de Processamento de Dados
(SERPRO)

---

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

---

Prof. Dr. Eduardo Cunha de Almeida
Universidade Federal do Paraná (UFPR)

Dedico este trabalho aos meus pais e irmão.

## AGRADECIMENTOS

Aos meus pais, por sempre acreditarem e investirem nos meus estudos. Ao meu irmão, por todo o incentivo e apoio incondicional. Obrigado por não desistirem de mim e estarem ao meu lado mesmo nas horas de cansaço.

Aos meus orientadores, Prof. Angelo Roncalli Alencar Brayner, Dr.-Ing e Prof. José de Aguiar Moraes Filho, Dr.-Ing por me aceitarem como aluno, pela confiança, paciência e sempre me motivaram a continuar mesmo num período difícil da minha vida como foi a pandemia. Obrigado, por todo o conhecimento compartilhado desde da época da minha graduação.

Agradeço a todos os meus amigos do grupo ARiDa: Arlino, Tiago, José Wellington, Salomão, Narciso, Lucas Cabral, Amanda, Matheus, Caio, Túlio, Daniel, Franklyn, Tibet e a todos que injustamente não foram mencionados, que de forma direta ou indireta sempre me apoiaram durante o mestrado. E é claro, ao Prof. Dr. José Maria da Silva Monteiro Filho por todas as lições e o seu entusiasmo, principalmente nos dias de aniversário e pizza. Vocês foram essenciais para a conclusão deste trabalho.

A todos os professores do programa de Mestrado e Doutorado em Ciência da Computação (MDCC) da UFC, pelo compromisso e aulas de altíssimo nível na minha formação enquanto aluno.

Ao CNPq e Funcap pela sua função fundamental no incentivo e apoio financeiro da pesquisa brasileira.

"Quando o interesse diminui, com a memória ocorre o mesmo."

(Johann Wolfgang von Goethe)

## RESUMO

Os sistemas gerenciadores de banco de dados (SGBD) nem sempre garantem que todos os dados estejam disponíveis na memória principal. SGBDs executam um algoritmo de substituição para determinar quais dados devem ser substituídos entre a memória principal e a mídia de armazenamento secundária. Essa classe de algoritmos está sempre em evolução e consideram cada vez mais novos aspectos nas decisões de substituição, como as mídias de armazenamento *flash* modernas. Muito provavelmente podemos estar iniciando uma nova era desses algoritmos com avanços extensivos em técnicas de predição. Este trabalho propõe o algoritmo EBRES (Efficient Buffer Replacement with Exponential Smoothing) que considera diversos aspectos em suas decisões e com o suporte de um modelo de predição simples de baixa sobrecarga para prever acessos a dados. Durante os experimentos, EBRES manteve um equilibro no seu desempenho em diferentes cenários testados e conseguiu proteger dados frequentes durante operações sequenciais.



**Palavras-chave:** bancos de dados; política de substituição de *buffer*; técnicas de predição.

## ABSTRACT

Database management systems (DBMS) do not always guarantee that all data is available in the main memory. DBMSs perform a replacement algorithm to determine which data to replace between main memory and secondary storage media. This class of algorithms is constantly evolving and increasingly considers new aspects in replacement decisions, such as modern *flash* storage media. Most probably, we may be starting a new era of these algorithms with extensive advances in forecasting techniques. This article proposes the EBRES (Efficient Buffer Replacement with Exponential Smoothing) algorithm that considers several aspects in the decisions with the support of a simple low-overhead forecasting model to predict data accesses. During the experiments, EBRES maintained a balance in its performance in different scenarios tested and was able to protect frequent data during sequential operations.



**Keywords:** databases; buffer replacement policy; forecasting techniques.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ACME | *Adaptive Caching Using Multiple Experts* |
| AD-LRU | *Adaptive Double LRU* |
| ARC | *Adaptive Replacement Cache* |
| BD | Banco de Dados |
| BPLRU | *Block-Level LRU* |
| CAR | *Clock with Adaptive Replacement* |
| CASA | *Cost-Aware Self-Adaptive* |
| CCCF-LRU | *Controllable Cold Clean First Least Recently Used* |
| CCF-LRU | *Cold-Clean-First LRU* |
| CFDC | *Clean-First Dirty-Clustered* |
| CFLRU | *Clean-First LRU* |
| CPU | *Central Processing Unit* |
| CRF | *Combined Recency and Frequency* |
| CRP | *Correlated Reference Period* |
| DLIRS | *Dynamic LIRS* |
| FAB | *Flash-Aware buffer* |
| FBR | *Frequency-Based Replacement* |
| FIFO | *First in First out* |
| GASA | *Ghost buffer Assisted and Self-tuning Algorithm* |
| H-ARC | *Hierarchical Adaptive Replacement Cache* |
| HDD | *hard disk drives* |
| IRR | *Inter-Reference Recency* |
| LeCaR | *Learning Cache Replacement* |
| LFU | *Least Frequently Used* |
| LFU-DA | *Least Frequently Used with Dynamic Aging* |
| LIRS | *Low Inter-reference Recency Set* |
| LLRU | *Locality-aware Least Recently Used* |
| LRFU | *Least Recently/Frequently Used* |
| LRU | *Least-Recently Used* |
| LRU-K | *Least kth-to-last Reference* |
| LRU-WSR | *LRU Write Sequence Reordering* |

| | |
|---|---|
| MQ | *Multi-Queue* |
| MRU | *Most-Recently Used* |
| OLTP | *Online Transaction Processing* |
| PA-LIRS | *Probability-based Adjustable algorithm on Low Inter-reference Recency Set* |
| RDF | *Resource Description Framework* |
| SGBD | Sistema Gerenciador de Banco de Dados |
| SPARQL | *SPARQL Protocol and RDF Query Language* |
| SQL | *Standard Query Language* |
| SSD | *solid-state drives* |

**SUMÁRIO**

# 1 INTRODUÇÃO

Esta dissertação se propõe a apresentar novas estratégias de substituição de *buffer* para Sistemas de Banco de Dados. Este capítulo introdutório apresenta uma contextualização na Seção 1.1; a Seção 1.2 motiva a pesquisa deste trabalho; a Seção 1.3 delineiam as hipóteses que guiam esta dissertação. O objetivo geral e os objetivos específicos são descritos na Seção 1.4, enquanto a metodologia deste trabalho é apresentada na Seção 1.5. Por fim, na Seção 1.7, define-se a estrutura desta dissertação.

## 1.1 Contextualização

Um Sistema Gerenciador de Banco de Dados (SGBD), em especial um SGBD relacional, armazena dados em blocos. Um bloco é armazenado em um arquivo do sistema de arquivos da máquina servidora do SGBD e é composto por uma sequência de dados. Em SGBDs relacionais, estes dados são chamados tuplas, e, portanto, um bloco contém um conjunto de tuplas. Quando o SGBD requisita um dado (tupla), ele pede ao sistema de arquivos o bloco específico no qual está armazenado a tupla requerida. Consequentemente, a unidade de transferência entre o sistema de arquivos e o SGBD é o bloco. O tamanho total armazenado em um banco de dados (tamanho da base) pode ser expresso pela quantidade total de blocos existentes.

No atual contexto tecnológico nem sempre é possível manter toda a base de dados e metadados na memória principal do servidor de banco de dados. O gerenciador de *buffer* é o componente de software de um SGBD responsável por solicitar blocos ao gerenciador de arquivos, com o objetivo de manter uma cópia desses blocos na memória principal, melhorando significativamente o desempenho de um banco de dados. Em uma implementação tradicional de um SGBD, o gerenciador de *buffer* pode ser visto como um componente central que conecta o processador de consulta e o gerenciador de arquivos.

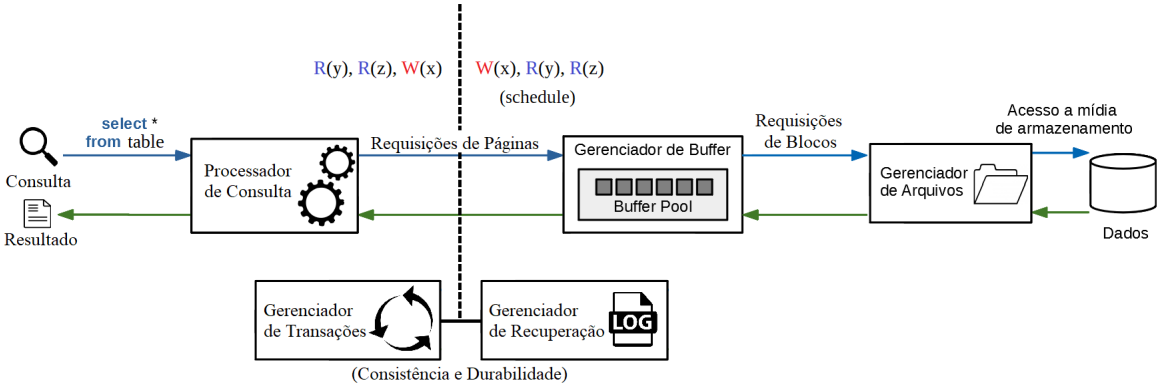
Os sistemas operacionais modernos fornecem um *buffer* implementado no sistema de arquivos. Se o SGBD utilizasse esse *buffer*, alguns problemas seriam percebidos, como por exemplo, em caso de falha, o SGBD não teria controle sobre o tempo e a ordem das escritas, afetando diretamente o processo de recuperação. Logo, é essencial usar um gerenciador de *buffer* específico para o SGBD.

SGBDs implementam um gerenciador de *buffer* para minimizar a diferença de desempenho entre as unidades de armazenamento. Apesar dos avanços tecnológicos, as hierarquias

de memória ainda são o principal modelo e impactam diretamente na arquitetura de um SGBD (EFFELSBERG; HäRDER, 1984). Podemos classificar as mídias de armazenamento em 2 (duas) classes: mídias simétricas e mídias assimétricas. Mídias simétricas são aquelas nas quais o custo e o tempo de leitura e gravação do dados é igual[1]. Mídias assimétricas, por sua vez, tem um custo de gravação superior ao de leitura, tornando, esta classe de mídia muito sensível às operações de gravação. Porém, como não tem componentes mecânicos em sua composição, apenas componentes eletromagnéticos – chips, as mídias assimétricas apresentam, no geral, um desempenho melhor que as simétricas.

Exemplos de mídia simétrica são os discos magnéticos ou *hard disk drives* (HDD). Enquanto que *solid-state drives* (SSD), e suas variantes como flash disk, memristor, NVDIM, etc., são considerados mídias assimétricas. Mesmo com a evolução dos componentes de hardware como os SSD, que apresentam um desempenho superior aos HDD, ainda há uma grande diferença, em termos de desempenho, entre essas mídias secundárias e a memória principal.

Figura 1 – Interações do gerenciador de *buffer* com outros componentes




Fonte: O Autor.

A Figura 1 descreve de forma simplificada o processo de consulta de dados de um SGBD fictício através das interações dos outros componentes com o gerenciador de *buffer*. Todavia, na prática, podem haver variantes deste processo, tendo mais ou menos componentes do que os apresentados[2]. Quando um SGBD é inicializado (processo de startup), ele pede ao sistema operacional uma quantidade de memória principal na qual ele, SGBD, alocará os blocos. Esta memória fornecida pelo sistema operacional ao SGBD é chamada de *Buffer Pool*.

[1] Na prática, os tempos de leitura e gravação em mídias simétricas não são estritamente iguais. Porém, a diferença é tão pequena, em torno de milésimos de microssegundo, que se torna desprezível.

[2] Isto é devido à própria arquitetura e implementação interna de um SGBD. Em qualquer caso, a Figura 1 representa uma abstração genérica e representativa das interações entre o gerenciador de *buffer* e os demais componentes de um SGBD.

Um bloco alocado (e residente) no *Buffer Pool* é chamado de *página*. A diferenciação é que na memória principal necessitamos de estruturas de controle sobre a página. Por exemplo, se foi modificada (ou não) desde o momento que foi trazida para o *Buffer Pool*, se foi (ou não) acessada recentemente, se sofreu (ou não) processo de recuperação por parte do gerenciador de recuperação, dentre outros.

O *Buffer Pool* é portanto um conjunto de páginas em memória principal onde, por sua vez, cada página corresponde a um (e somente um) bloco vindo do sistema de arquivos, com a adição de estruturas de controle do *Buffer Pool*.

Quando uma consulta (por exemplo, na linguagem SQL ) é enviada, o processador de consultas executa as etapas de processamento da consulta. Durante a execução, uma série de solicitações de página é emitida para o gerenciador de *buffer*. Caso as páginas não estejam no *Buffer Pool*, o gerenciador de *buffer* solicita blocos ao gerenciador de arquivos, acessando a mídia física de armazenamento. Além disso, os componentes de gerenciamento de transação e recuperação devem garantir a consistência e durabilidade do banco de dados durante as requisições feitas pelo processador de consultas ao gerenciador de *buffer* (HELLERSTEIN *et al.*, 2007).

Em resumo, o gerenciador de *buffer* propõe minimizar o custo do acesso físico (E/S) à mídia de armazenamento secundária[3] com a ajuda de outros componentes. Um dos pontos a otimizar dentro do gerenciador de *buffer* são as políticas de substituição de páginas, responsáveis por determinar quais páginas devem residir na memória principal. Quando não há memória suficiente no *Buffer Pool* para alocar uma nova página, o gerenciador de *buffer* consulta a política de substituição que escolhe uma página *P* (no *Buffer Pool*) como vítima (evicção), removendo *P* da memória principal, gravando esta página de volta na mídia secundária (se a página tiver sido modificada anteriormente), pedindo ao gerenciador de arquivos o bloco requisitado, e alocando a nova página no *Buffer Pool* para o referido bloco.

Uma política de substituição de páginas inteligente tenta garantir que na maioria das requisições de páginas, a página requisitada já esteja alocada no *Buffer Pool*, sem a necessidade de evicção. Isto é chamado taxa de acerto (ou hit ratio ou simplesmente hit), e é um dos indicadores de desempenho de um gerenciador de *buffer*. O caso contrário, no qual a evicção ocorre, é chamado de taxa de não-acerto (ou miss). Note que em mídias assimétricas, uma evicção pode disparar mais de uma gravação física na mídia (JO *et al.*, 2006). Em qualquer caso, o processo

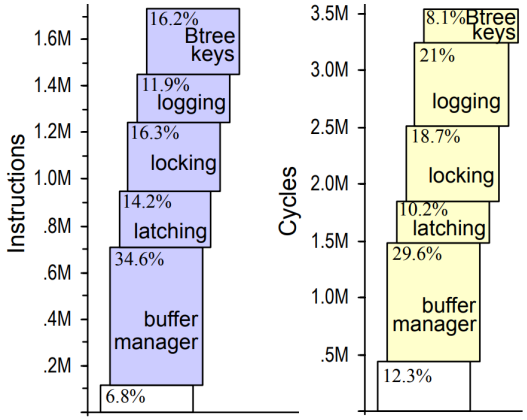
[3] Este acesso físico é também chamado de E/S – Entrada/Saída, ou em inglês, I/O – Input/Output. Ao longo da dissertação usaremos o termo simplificado E/S para indicar acesso à mídia secundária.

de evicção implica acesso à mídia secundária, seja para a leitura e/ou para a gravação, o que, devido à baixa velocidade da mídia secundária comparada à da memória principal, contribui para o baixo desempenho do SGBD. Notadamente, para um bom desempenho, espera-se que a taxa de acerto tenda a 1 (100%)[4].

## 1.2 Motivação

Em (HARIZOPOULOS *et al.*, 2008), encontra-se evidência de que o componente gerenciador de *buffer* de um SGBD apresenta um maior consumo de CPU dentre diversos componentes de um SGBD. A Figura 1 mostra o número de instruções e ciclos de CPU realizadas pelos principais componentes de um SGBD ao executar a transação *New Order* com características OLTP do *benchmark* TPC-C (TPC, 2010). Os quadros brancos dos gráficos de instruções e ciclos (com valor de 6,8% e 12,3%, respectivamente) representam o trabalho real da execução da consulta. O gerenciador de *buffer* possui um impacto maior do que os demais componentes devido a criação, busca e acesso de dados. Realizar otimizações nesse componente pode trazer ganhos significativos de desempenho ao SGBD.

Figura 2 – Impacto de uma consulta OLTP aos componentes de um SGBD




Fonte: Harizopoulos *et al.* (2008).

Compreende-se que a política de substituição pode ser um dos pontos de otimização do gerenciador de *buffer*, pois dependendo do comportamento deste componente podemos reduzir ou aumentar o consumo de CPU. Durante a execução de consultas é comum a geração de diferentes planos de execução que podem gerar diferentes padrões de acesso que resultam em requisições de páginas à política de substituição. Dessa forma, a carga de trabalho de um SGBD é dinâmica, pois é dependente do usuário e aplicações. Os padrões de acesso impactam diretamente

[4] De fato, pela natureza do problema, a taxa de acerto nunca atinge 100%, mas deve se manter perto deste número.

no desempenho. Por exemplo, um padrão de leitura sequencial pode ocupar toda a memória rapidamente, removendo dados frequentes que são importantes durante o processamento. Em síntese, elaborar estratégias de substituição adaptativas aos padrões de acesso são essenciais para manter um bom desempenho ao reduzirem o acesso à mídia de armazenamento secundária e consequentemente o consumo de CPU (KABRA; DEWITT, 1998; MEGIDDO; MODHA, 2003).

## 1.3 Hipóteses

Com a evidencia de que o gerenciador de *buffer* corresponde a cerca de 1/3 dos ciclos de CPU dentre todos os componentes de um SGBD (HARIZOPOULOS *et al.*, 2008), e que a política de substituição de páginas é o "núcleo"de um gerenciador de *buffer*, e, portanto, sensível a qualquer melhoria, fizemos uma pesquisa bibliográfica apresentada sobretudo nas Seções 3.1 e 3.2 que nos levaram a elaboração das seguintes hipóteses:

- **Hipótese 01:** SGBDs que implementam estratégias de substituição com uma complexidade computacional constante tendem a apresentar um melhor desempenho.
- **Hipótese 02:** Estratégias de substituição que tiram proveito das tecnologias de armazenamento assimétricas quando buscam reduzir o número de escritas podem fornecer uma melhoria significativa no desempenho.

## 1.4 Objetivo

O Objetivo geral desse trabalho é a elaborar de uma estratégia de substituição de páginas de *buffer* que usa técnicas de predição com uma complexidade constante, e tirando proveito da assimetria da mídia de armazenamento, ser capaz de manter um equilíbrio entre o número de escritas e a taxa de acerto. Para os objetivos específicos temos:

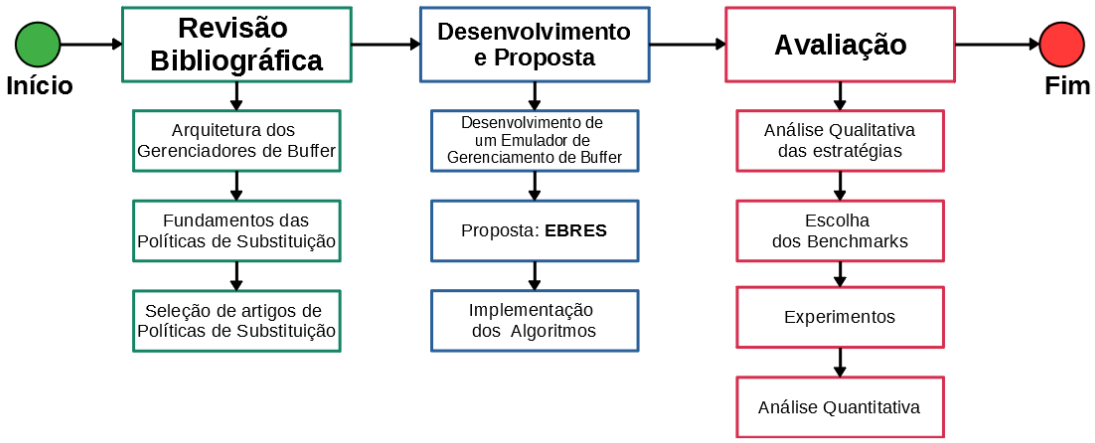
- Realizar um levantamento biográfico e descrever o funcionamento de algumas das principais estratégias de substituição.
- Realizar uma análise comparativa qualitativa de algumas dessas estratégias apresentadas que mais se destacam e são fundamentais para a elaboração do algoritmo proposto.
- Desenvolver uma nova política de substituição de páginas para o gerenciador de *buffer* de um SGBD chamada EBRES (*Efficiente Buffer Replacemente wit Exponential Smoothig*).
- Executar testes comparativos entre EBRES e outras políticas publicadas na literatura, com

diferentes cargas de trabalho, e avaliar o desempenho dos algoritmos propostos.

## 1.5 Metodologia

A Figura 3 apresenta a metodologia usada na concepção deste trabalho aplicado, uma vez que essa pesquisa gera conhecimentos (dentre eles, um artefato tecnológico) para aplicações práticas e dirigidas à solução de problemas específicos (BARROS; LEHFELD, 2000). Para obter os objetivos apresentados na Seção 1.4, a pesquisa foi dividida em três etapas, sendo a primeira a revisão biográfica, a segunda o desenvolvimento e proposta, e a terceira a avaliação empírica quantitativa.

Figura 3 – Metodologia do trabalho




Fonte: O Autor.

Primeiramente, foi realizado o planejamento de uma revisão biográfica com um estudo das arquiteturas dos gerenciadores de *buffer*, trabalhos como (EFFELSBERG; HäRDER, 1984) e (HELLERSTEIN *et al.*, 2007) abrangem uma compreensão avançada sobre a implementação deste componente. Em seguida, foi feito um estudo sobres os principais fundamentos das políticas de substituição de *buffer*, trabalhos como (O'NEIL *et al.*, 1993), (MEGIDDO; MODHA, 2003) e (PARK *et al.*, 2011), trazem noções avançadas de novos mecanismos das suas respectivas épocas e nos ajudam a entender vários destes fundamentos, por exemplo a influência das memórias *flash* no campo das políticas de substituição. Por fim, foram selecionados diversos artigos relevantes de políticas de substituição, sendo eles majoritariamente citados nas Seções 3.1 e 3.2.

Na etapa de desenvolvimento e proposta, inicialmente foi implementado um geren-

ciador de arquivos e gerenciador de *buffer* (ver Seção 5.1) em linguagem C, capaz de realizar chamadas diretas ao sistema operacional e emular o processamento das transações. Em seguida foi elaborado uma proposta de uma nova política de substituição de *buffer* denominada EBRES (ver Seção 4). Por fim, as principais estratégias das Seções 3.1 e 3.2 foram implementadas no gerenciador de *buffer*.

Na etapa de avaliação, inicialmente foi realizado uma análise comparativa qualitativa das principais estratégias das Seções 3.1 e 3.2 destacando as suas principais características. Em seguida, foi escolhido uma série de *benchmarks* com cargas de trabalho reais e sintéticas que foram usados para execução de experimentos nas políticas de substituição com o objetivo de realizar uma análise comparativa quantitativa.

## 1.6 Contribuições

As principais contribuições desta dissertação são as seguintes:

- Um estudo sobre diversos algoritmos de substituição de *buffer* relevantes para o contexto de banco de dados.
- Uma nova política de substituição de *buffer* denominada EBRES.
- O desenvolvimento de um gerenciador de arquivos e gerenciador de *buffer* usado para avaliação experimental.
- A implementação e um estudo comparativo entre os principais algoritmos de substituição de *buffer*.

## 1.7 Organização da dissertação

Os capítulos da dissertação estão organizados da seguinte forma:

- Capítulo 2: apresenta a fundamentação teórica do trabalho proposto.
- Capítulo 3: reúne diversos trabalhos relacionados das principais estratégias de substituição na literatura
- Capítulo 4: descreve a solução proposta, a política de substituição EBRES.
- Capítulo 5: apresenta a avaliação experimental das estratégias.
- Capítulo 6: apresenta as conclusões, e perspectivas de continuação deste trabalho.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica de gerenciamento de *buffer* em SGBD. A Seção 2.1 apresenta uma visão geral da arquitetura dos gerenciadores de *buffer* de banco de dados. A Seção 2.2 discorre sobre alguns dos principais fundamentos das políticas de substituição de *buffer*. Problemas inerentes do gerenciamento de *buffer* em SGBDs são descritos na Seção 2.3. O impacto do uso de mídias assimétricas para o gerenciador de *buffer* é estudado na Seção 2.4. A Seção 2.5 brevemente descreve um algoritmo de substituição de referência, que por ser ótimo, serve como parâmetros de comparação em avaliação empírica de políticas de substituição de *buffer*. Por fim, a Seção 2.6 conclui este capítulo.

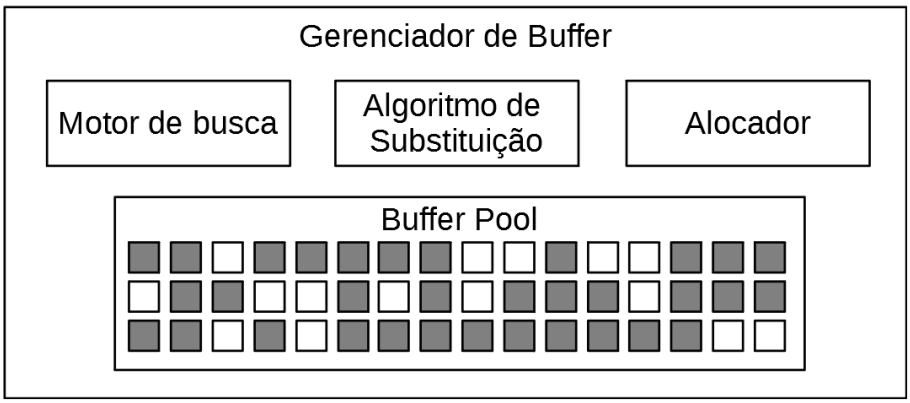
## 2.1 Arquitetura dos gerenciadores de *buffer*

Os SGBDs certamente são um dos sistemas de software mais complexos, pois devem implementar diversos requisitos funcionais e não funcionais para atender usuários e aplicações. Como discutido na Seção 1.1, SGBDs são implementados com diversos componentes que trabalham juntos para prover tais requisitos. Dentre estes componentes, encontra-se o gerenciador de *buffer* que pode apresentar um elevado custo computacional em relação aos demais componentes (ver Seção 1.2).

O gerenciador de *buffer* é o componente de software responsável por solicitar blocos ao gerenciador de arquivos para manter uma cópia desses blocos na memória principal (chamada página), melhorando significativamente o desempenho do SGBD. No contexto tecnológico atual, ainda há uma grande diferença do tempo de acesso aos dados entre a memória principal e as mídias de armazenamento secundárias, ficando evidente que, para um SGBD tradicional atingir um desempenho ideal, deve ser maximizado o número de requisições de página atendidas pelo gerenciador de *buffer* sem a necessidade de acessos aos dispositivos de armazenamento secundários. Como nem sempre é possível manter todas as páginas do banco de dados disponíveis na memória principal, também é função do gerenciador de *buffer* determinar qual o melhor subconjunto de páginas que deve residir na memória principal para atender as requisições de maneira eficiente.

A Figura 4 apresenta alguns subcomponentes do gerenciador de *buffer*. O subcomponente motor de busca é responsável por encontrar as páginas solicitadas pelos componentes de alto nível. Uma estratégia simples seria realizar uma busca sequencial para todas as páginas que

Figura 4 – Subcomponentes do Gerenciador de *buffer*




Fonte: O Autor.

estão atualmente alocadas no *buffer*, mas o custo e a frequência dessa operação podem ter um impacto negativo significativo no desempenho do SGBD. Portanto, realizar pesquisas usando tabelas de hash ou outros algoritmos de indexação são boas alternativas para encontrar as páginas solicitadas (EFFELSBERG; HäRDER, 1984).

Quando o SGBD inicializa o gerenciador de *buffer* solicita um espaço de memória principal para alocar páginas através do sistema operacional. Este espaço é chamado de *Buffer Pool*. Em termos práticos de implementação, é o espaço de memória alocado para páginas de dados, geralmente é um valor fixo. Esse espaço também pode ter limites inferiores ou superiores, portanto, a alocação de memória das páginas deve variar apenas dentro desses limites. Na Figura 4, o Alocador representa uma série de funções responsáveis por inicializar e manipular as estruturas de dados utilizadas no gerenciador de *buffer* e alocar os dados no *Buffer Pool*.

O *Buffer Pool* é implementado usando uma matriz de *frames* (região física da memória para alocação de blocos) onde cada *frames* é preferencialmente do tamanho de um bloco de mídia de armazenamento. Associado à matriz de *frames*, temos o subcomponente motor de busca que mapeia todos os blocos e indica a localização de seus respectivos frames no *Buffer Pool*. Portanto, as páginas são os frames (com seus respectivos dados de blocos) e com descritores que fornecem metadados sobre os dados alocados nos *frames*, como, por exemplo, o *dirty bit*, que indica que a página foi modificada (HELLERSTEIN *et al.*, 2007).

Também é importante notar que existem dois tipos de páginas, leitura *(clean)* ou escrita *(dirty)*. Quando os outros componentes do SGBD requisitam páginas do gerenciador de *buffer*, eles indicam que tipo de operação eles executarão. Um bloco só é gravado de volta na mídia de armazenamento se uma operação de escrita o modificou no *Buffer Pool* (i.e., em memória principal). O gerenciador de *buffer* fornece um serviço de propagação de dados

local para componentes de nível superior, como o processador de consultas. Uma página pode ser reutilizada indefinidamente por vários aplicativos sem sempre buscá-la na mídia de armazenamento para cada acesso solicitado. Isto é chamado de reentrância de dados. Por esse motivo, o gerenciador de *buffer* melhora o desempenho do SGBD.

Outra vantagem do gerenciador de *buffer* é que os componentes de nível superior de um SGBD não precisam necessariamente saber se a página está na memória ou armazenada em mídia física. Isso simplifica o trabalho desses componentes, pois eles podem assumir que todo o banco de dados está na memória e precisam apenas solicitar páginas com suas operações. Por exemplo, se o processador de consultas precisa fazer uma varredura completa em uma tabela (operações de leitura de dados) ele pode consultar os metadados do catálogo para saber qual ID de página (identificador de página) será solicitado ao gerenciador de *buffer*. Dentre as informações usadas por estes componentes de nível superior temos (GARCIA-MOLINA *et al.*, 2001):

- Dados: o conteúdo armazenado no banco de dados por aplicações e usuários.
- Metadados: as informações que descrevem a estrutura do banco de dados e restrições sobre ele.
- Estatísticas: as informações obtidas pelo SGBD sobre propriedades específicas dos dados.
- Índices: as estruturas de dados que provem um acesso mais eficiente as informações do banco de dados.

O *Buffer Pool* do gerenciador de *buffer* tem um número limitado de páginas. Como resultado, pode haver solicitações de páginas que não estejam na memória principal. Suponha que uma página específica já esteja na memória *(hit)*. É um cenário perfeito, pois não há custo de acesso à mídia de armazenamento físico para encontrá-la. No entanto, suponha que a página não esteja na memória *(miss)*. Nesse caso, o gerenciador de *buffer* solicita os dados do componente gerenciador de arquivos e coloca a página no *Buffer Pool*, tendo um alto custo de acesso aos dados na mídia de armazenamento secundária.

O gerenciador de *buffer* usa o gerenciador de arquivos para mover blocos da mídia de armazenamento secundário para o *Buffer Pool*. No entanto, suponha que o *Buffer Pool* esteja cheio, sem espaço suficiente para alocar a página solicitada. Nesse caso, o gerenciador de *buffer* utiliza um recurso chamado política de substituição de página, que tem como objetivo determinar quais páginas permanecem ou são desalocadas (despejadas) da memória. O subcomponente algoritmo de substituição escolhe uma vítima com base em estratégias que determinam a página

com o valor mais baixo, no nível do *buffer*, atualmente em memória. As estratégias de tomada de decisão de vítimas são baseadas no princípio da localidade, que se divide em localidade temporal e espacial.

- Localidade espacial: Na maioria das vezes, sempre que possível, é vantajoso armazenar sequencialmente blocos em um mesmo arquivo na mídia de armazenamento, pois isso facilita os mecanismos de hardware de busca de blocos. Por exemplo, ao solicitar a leitura completa deste arquivo específico, implica uma leitura sequencial de cada bloco. A localidade espacial demonstra maior probabilidade de acessar dados próximos aos dados solicitados. Assim, após um bloco ser acessado, há uma alta probabilidade de que o próximo bloco solicitado seja seu vizinho armazenado fisicamente.
- Localidade temporal: Diz respeito ao uso frequente de uma página/bloco. Quando um bloco específico é acessado várias vezes, há uma probabilidade maior de ser acessado novamente. É típico que um processador de consultas tenha algoritmos que possam acessar as mesmas páginas repetidamente em um curto período, portanto, é uma vantagem manter essas páginas frequentes na memória principal.

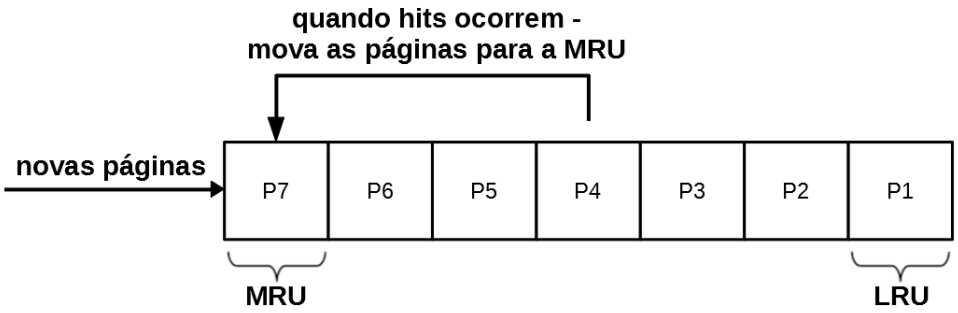
Alguns algoritmos de substituição de página consideram apenas a localidade temporal, tendo a frequência de acesso como seu principal recurso na sua tomada de decisões, isso pode agravar problemas quando um comportamento de acesso segue o princípio da localidade espacial (ver Seção 2.3.1).

Os algoritmos de substituição podem tirar proveito da técnica de pré-paginação que consistem em ler conjunto de blocos armazenados fisicamente próximos na mídia secundaria em vez de ler apenas um bloco por vez, aproveitando ao máximo a largura de banda de acesso do dispositivo (EFFELSBERG; HäRDER, 1984). Pré-paginação favorece o padrão de acesso que tende a seguir o princípio da localidade espacial, já que blocos pré-carregados já estão previamente no *Buffer Pool* esperando para serem requisitados. Porém, existe uma desvantagem quando temos um padrão de acesso que tende a seguir o princípio da localidade temporal, a probabilidade que vários blocos pré-carregados não sejam usados pelas transações é muito alta ocupando espaço no *Buffer Pool*. O uso ou não da técnica de pré-paginação afeta diretamente a performance do algoritmo de substituição de página. Isso gera uma decisão arquitetural durante a implementação de um gerenciador de *buffer*.

## 2.2 Visão geral das políticas de substituição de *buffer*

Para construir um algoritmo de substituição de páginas eficiente é essencial conhecer as operações do banco de dados, tanto as operações realizadas no momento da solicitação quanto as que podem vir. Uma estratégia ideal para todos os cenários é desconhecida. Vários fatores internos e subjacentes ao gerenciador de *buffer* podem influenciar a ordem e o entrelaçamento das operações (SILBERSCHATZ *et al.*, 2005). Em qualquer caso, uma política de substituição adequada deve maximizar o reuso de dados, para que, na grande maioria das vezes que uma página é solicitada ela já resida no *Buffer Pool* (na memória). A seguir, apresentamos algumas das políticas básicas utilizadas na construção de políticas sofisticadas.

Figura 5 – Fluxo de execução da LRU e MRU




Fonte: O Autor.

*Least-Recently Used* (LRU) e *Most-Recently Used* (MRU) são políticas de substituição de página que podem ser implementadas usando uma lista, seguindo a ordem das referências. A figura 5 demonstra o fluxo de execução. Quando uma página é acessada, ela é movida para o topo da lista (MRU), indicando que foi a última página referenciada. A mesma coisa acontece quando uma página é nova. Denominamos referência de página, a requisição, com o acesso aos seus dados para processamento, de uma página, esteja ela ou não em memória.

As páginas menos recentes tendem a estar no final da lista (LRU), então quando o *Buffer Pool* está cheio e não tem memória suficiente para alocar novas páginas, podemos optar por remover a página menos recente (estratégia LRU) ou a página mais recente (estratégia MRU) que está no topo da lista.

*Least Frequently Used* (LFU) é uma política de substituição que mantém um contador de referências para cada página. Este contador indicará qual frequência esta página é acessada, incrementando o contador a cada nova referência. A página com o menor contador (página menos frequente) será escolhida como vítima. Existem diversas variações desta política, e ela pode ser

implementada de várias maneiras, por exemplo, com uma lista ordenada pela frequência de cada página. A LFU apresenta algumas desvantagens, como por exemplo, a complexidade de manter a ordenação das páginas e também o fato que o algoritmo poder preservar uma página frequente por muito tempo mesmo que essa página não tenha sido referenciada posteriormente, devido à páginas com os contadores com uma alto valor. Para mitigar este problema, existem estratégias para envelhecer as páginas *(aging)*. Por exemplo, *Least Frequently Used with Dynamic Aging* (LFU-DA) (ARLITT *et al.*, 2000) usa o fator de inflação $L$ inicializado em 0. O valor de $L$ é sempre atualizado com o valor do contador da última vítima. Quando uma nova página é inserida, o contador de frequência é adicionado com o valor atual de $L$, diminuindo o esforço para que as novas páginas possam aumentar a sua frequência antes de serem selecionadas como vítimas.

*First in First out* (FIFO) é uma política de substituição simples que usa uma fila de páginas. A inserção de novas páginas acontece no início da fila. Quando uma substituição é necessária, a página no final da fila é removida. Ao contrário de LRU e MRU, o algoritmo FIFO não move nenhuma página quando elas são referenciadas novamente.

## 2.3 Problemas Inerentes ao Gerenciamento de *buffer*

Adicionalmente aos mecanismos básicos de gerenciamento de *buffer* visto na seção anterior, existem problemas inerentes ao processo de gerenciamento de *buffer*. Os métodos de gerenciamento de *buffer* (políticas de substituição) podem endereçar total ou parcialmente estes problemas.

### *2.3.1 Operações Sequenciais e Scan resistance*

Um dos recursos essenciais de uma política de substituição sofisticada é o *scan resistance*, que coíbe que o *buffer* seja todo poluído com páginas recentes, possivelmente, removendo páginas mais referenciadas.

É uma operação relativamente comum quando o processador de consultas executa a operação sequencial (por exemplo, uma varredura completa na tabela ou índice – *sequential scan*). Várias dessas páginas recentes possivelmente nunca serão referenciadas novamente durante a execução da aplicação, e talvez nem mesmo depois, em futuro próximo. Operações sequenciais podem remover páginas frequentes ou importantes dentro do contexto de execução do gerenciador de *buffer* artificializando o reuso das páginas. Algoritmos como o ARC (ver

Seção 3.1.9) (MEGIDDO; MODHA, 2003) usam mecanismos de *scan resistance* para reduzir o impacto dessas operações sequenciais.

SGBS comerciais vem tentando minimizar a quantidade de dados que são trazidos para o *Buffer Pool* e dificilmente serão acessados, o objetivo é manter as páginas acessadas com uma maior frequência no *Buffer Pool* mesmo que um longo acesso sequencial seja executado. Para isso, SGBDs como o ORACLE e MYSQL optam em utilizar uma estratégia de inserção de páginas recentes mais sofisticada (ver Seção 3.1.3).

### *2.3.2 Cache Starvation*

Várias políticas de substituição dividem o *buffer* em regiões (geralmente listas) com tamanhos fixos ou dinâmicos. Cada região pode manter páginas recentes, frequentes, *clean* ou *dirty*, ou combinações desses tipos. Essas regiões competem por uma fração do *buffer* e dependendo da carga de trabalho, uma dessas regiões será favorecida e seu tamanho será mais significativo. Consequentemente, as outras regiões se tornarão menores.

O problema de *cache starvation* ocorre quando uma região se torna muito pequena e os mecanismos da política de substituição impedem que ela cresça novamente, possivelmente levando à degradação do desempenho do algoritmo. Evitar o *cache starvation* é uma qualidade essencial no desenvolvimento de novas políticas de substituição.

### *2.3.3 Páginas fantasmas (ghost)*

No projeto de algoritmos de substituição, quando uma página é removida do *buffer*, o espaço reservado para dados é removido ou substituído da memória dependendo da implementação. No entanto, uma abordagem comum é manter apenas alguns metadados dessa página despejada na memória, por exemplo, o identificador de página. Assim, o algoritmo pode fazer suposições sobre essa página despejada caso ela seja acessada novamente no futuro. O algoritmo 2Q (ver Seção 3.1.5) usa esta abordagem para identificar *hot pages*, que são páginas consideradas importantes para o algoritmo de substituição durante o seu contexto de execução (SHASHA; JOHNSON, 1994).

O algoritmo LIRS (ver Seção 3.1.8), publicado anos depois de 3.1.5, usou essa abordagem de manter *ghost pages* chamando pelo termo *non-resident pages*. Posteriormente, o algoritmo ARC (ver Seção 3.1.9) usou o termo *ghost caches*. Neste trabalho, usaremos o termo *ghost pages* para explicar os algoritmos.

## 2.4 Mídia de armazenamento assimétrica

Vários algoritmos de substituição foram desenvolvidos ao longo dos anos considerando aspectos de escolha de vítima como a frequência, recência e falhas de página *(miss)*. Porém, com a evolução das novas tecnologias de armazenamento e a popularização dos SSDs, um novo aspecto surgiu e teve que ser considerado no desenvolvimento de políticas de substituição de páginas sofisticadas.

As unidades de disco rígido HDD exigem o mesmo tempo e custo para operações de leitura e gravação. Portanto, o acesso aos dados é feito de forma simétrica. Por outro lado, os SSDs possuem um acesso à mídia de forma assimétrica. O tempo e o custo das operações de gravação em um SSD são mais significativos do que as operações de leitura (PARK *et al.*, 2011).

Consequentemente, para que uma estratégia de substituição se beneficie da assimetria, ela deve ter mecanismos para diferenciar as páginas entre leitura e escrita. Se o algoritmo conseguir reduzir o número de páginas de escrita despejadas, é esperado que o tempo e o custo das transações serão menores, melhorando assim o desempenho.

O algoritmo *Clean-First LRU* (ver Seção 3.2.1) foi a primeira estratégia desenvolvida para trabalhar com SSD (memória flash). Várias novas estratégias foram desenvolvidas nos anos seguintes motivadas pela CFLRU e novas mídias de armazenamento assimétricas também surgiram. Portanto, tirar proveito da assimetria da mídia de armazenamento é um recurso essencial no desenvolvimento de novas políticas de substituição.

## 2.5 Algoritmo MIN

O algoritmo MIN é uma estratégia offline, pois a política não é executada em um sistema real de produção, pois é necessário conhecimento futuro da carga de trabalho para a sua execução. A implementação pode usar uma lista de páginas residentes na memória e uma fila com todas as referências de página para o futuro, a ideia básica por trás do processo de substituição de página é escolher uma vítima na lista com menor frequência de uso no futuro (Belady, 1966).

MIN é considerado um algoritmo ótimo e pode maximizar a taxa de acerto *(hit ratio)* de um *buffer* com um número fixo de páginas. Por ser um algoritmo offline, ele é usado para realizar testes de desempenho e comparação empírica para encontrar a melhor taxa de acertos para uma determinada carga de trabalho.

## 2.6 Conclusão

Este capítulo reviu a arquitetura e problemas enfrentados pelo gerenciador de *buffer*, em especial na sua política de substituição de páginas. Estudou também o impacto de novas mídias assimétricas de armazenamento no gerenciamento de *buffer* em SGBDs.

# 3 TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos relacionados estudados durante a revisão bibliográfica. Eles foram classificados em duas categorias. A primeira está descrita na Seção 3.1 que apresenta as políticas de substituição de *buffer* do tipo não assimétricas. A segunda na Seção 3.2 com as as políticas de substituição de *buffer* assimétricas. Em seguida, a Seção 3.3 faz uma breve compilação no formato de uma linha do tempo com o objetivo de revelar possíveis relações entre trabalhos apresentados. Por fim, a Seção 3.4, faz uma análise qualitativa e comparativa dos trabalhos relacionados discutidos nas Seções 3.1 e 3.2.

## 3.1 Políticas de substituição de *buffer* não assimétricas

Esta seção apresenta as políticas de substituição de *buffer* projetadas para mídias não assimétricas. Foram analisados trabalhos relevantes na literatura e identificamos seus potenciais problemas. Essas políticas focam na melhoria da taxa de acertos, embora não considerem o tipo de operação (leitura/gravação). Mecanismos de histórico e envelhecimento sofisticados têm sido propostos. Estes conceitos melhoram a classificação das páginas frequentes e se adaptam melhor aos padrões de acesso.

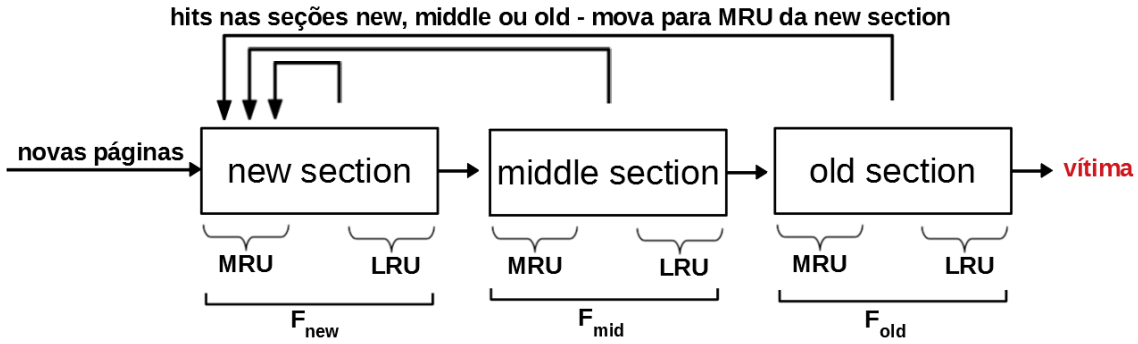
### *3.1.1 FBR*

*Frequency-Based Replacement* (FBR) foi um dos primeiros algoritmos a combinar aspectos de recência e frequência usando os algoritmos LRU e LFU como base. O FBR divide o *buffer* em três seções: *new*, *middle*, *old*. Cada página tem um contador de frequência semelhante ao LFU. O FBR tem dois parâmetros de ajuste: $A_{max}$ e $C_{max}$. O parâmetro $A_{max}$ é usado para evitar que os contadores de referência cresçam indefinidamente e o $C_{max}$ é o valor máximo de entidades usadas na estratégia de busca de vítimas (ROBINSON; DEVARAKONDA, 1990).

A Figura 6 descreve o fluxo de execução simplificado do FBR. Os parâmetros de ajuste $F_{new}$ e $F_{old}$ definem o tamanho das seções ($F_{mid}$ é o resto). As novas páginas são inseridas na MRU da seção *new* com a frequência igual a 1. Se a seção *new* estiver cheia, a página LRU da seção *new* é movida para a seção *middle*. Se a seção *middle* também estiver cheia, a página LRU da seção *middle* então será movida para a seção *old*. E, por último, se a seção *old* estiver cheia, o algoritmo escolhe a página com a menor contagem de referência da seção *old* como vítima.

Quando ocorre um *hit*, as páginas são movidas para a MRU da seção *new* e o contador

Figura 6 – FBR fluxo de execução




Fonte: O Autor.

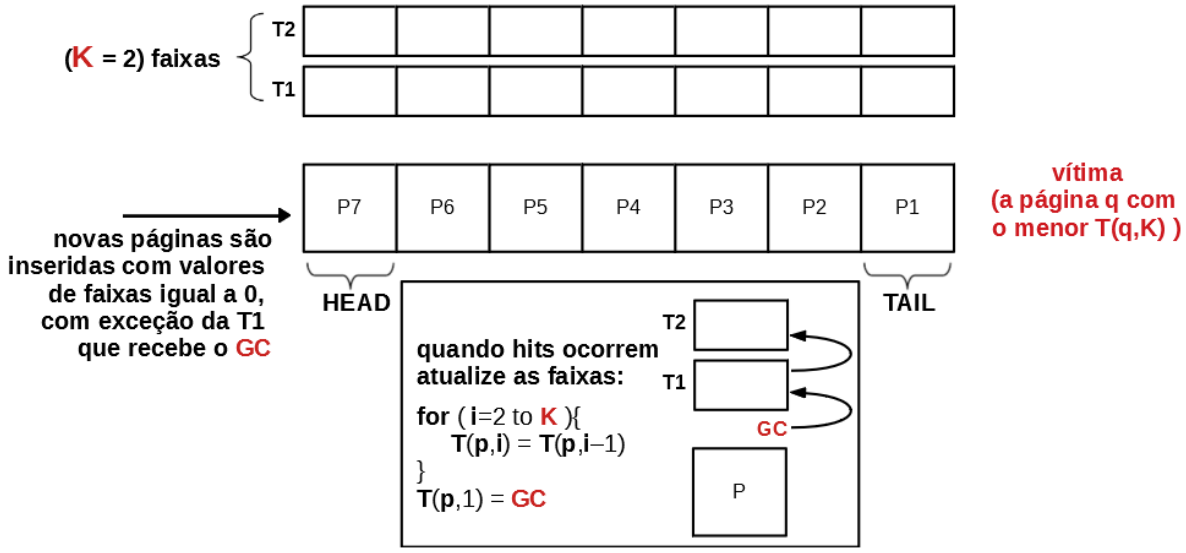
de frequência da página é incrementado. No entanto, o contador não será incrementado se a página já residia na seção *new*. Essa estratégia evita que páginas acessadas com frequência em curtos períodos tenham contadores de referência muito grandes.

A escolha dos tamanhos das seções do FBR influencia na sua performance. Quanto maior o tamanho da seção *new*, maior será o número de páginas recentes e maior será o tempo de vida em que uma página reside no *buffer*, pois devem primeiramente envelhecerem até serem deslocadas para as nas seções *middle* e *old*. Quanto maior o tamanho da seção *middle*, maior será a probabilidade de uma página ser referenciada novamente sem correr o risco no pior caso (*miss*) de ter sido movida para a seção *old* e selecionada como vítima anteriormente. Quanto maior o tamanho da seção *old*, durante a escolha da vítima mais páginas com diferentes contadores de referência podem ser consultados.

### *3.1.2 LRU-K*

O algoritmo LRU não discrimina com sucesso a frequência das páginas, pois só pode considera a última referência por página. O algoritmo LFU discrimina a frequência, mas tem a desvantagem de manter as páginas com alta frequência por muito tempo no *buffer* mesmo que não sejam novamente referenciadas. Portanto, é necessário uma estratégia de envelhecimento, por exemplo, decrementos de contador de referência. O *Least kth-to-last Reference* (LRU-K) (O'NEIL *et al.*, 1993) melhora o algoritmo LRU mantendo um histórico de faixas *K*, com o tempo das últimas *K* referências relacionadas a cada página no *buffer*. O *K* representa um parâmetro de ajuste. Quando *K* é igual a 1 (LRU-1), o algoritmo funciona como um LRU simples. Ao contrário do LFU, o algoritmo aplica uma estratégia sofisticada de envelhecimento sem ter que diminuir constantemente os contadores de referência.

Figura 7 – LRU-K fluxo de execução




Fonte: O Autor.

A Figura 7 descreve o fluxo de execução simplificado do LRU-K. O algoritmo usa um contador global *(GC)* que é incrementado para cada solicitação e outro parâmetro de ajuste chamado *Correlated Reference Period* (CRP). CRP é uma estratégia otimista que mantém a página no *buffer* por um curto período, evitando que ela seja imediatamente escolhida como vítima.

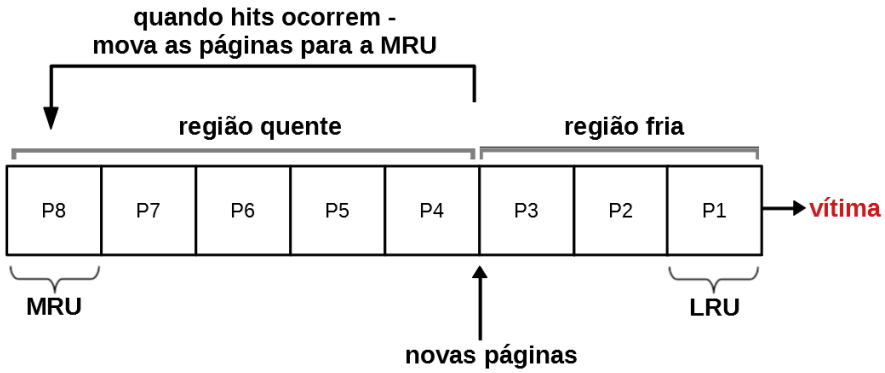
Neste exemplo, vamos supor que CRP é igual a 0 e *K* é igual a 2 (LRU-2), para simplificar a explicação do algoritmo. Novas páginas são inseridas em uma fila de prioridade e cada faixa associada a essa nova página é definida como 0, exceto a faixa 1 (T1), que é definida como o valor atual do contador global. Quando ocorre um *hit*, os valores de cada faixa devem ser atualizados. Em primeiro lugar, os valores das faixas da página referenciada são deslocados. Neste caso (LRU-2), a última referência em T1 é definida como T2. Finalmente, T1 é definido como o valor atual do *GC*.

A escolha da vítima é feita selecionando uma página com o menor valor na faixa *K*. Neste caso (LRU-2), a página com o menor valor na faixa T2. Para encontrar a vítima com o menor valor, o LRU-K usa uma fila de prioridade, que pode ser implementada com uma estrutura de heap, por exemplo. Assim, a complexidade do algoritmo é O(log N). Como as faixas de uma página do histórico são atualizadas quando ocorrem *hits*, as referências mais antigas são descartadas e consequentemente não serão mais usadas para selecionar uma vítima isso permite que páginas anteriormente consideradas frequentes sejam envelhecidas a medida que a carga de trabalho muda.

### *3.1.3 LRU-MIS (LRU with Midpoint Insertion Strategy)*

O algoritmo LRU simples não possui *scan resistance* (ver Seção 2.3.1). Uma possível solução para essa vulnerabilidade é a estratégia de inserção de ponto médio. A Figura 8 descreve o fluxo de execução do LRU-MIS. A diferença para a LRU simples é que as novas páginas são inseridas em um ponto médio na lista, dividindo a LRU em duas sublistas, regiões quentes e frias. Portanto, as páginas são inseridas no lado da MRU somente quando ocorre um *hit*. A escolha da vítima é a mesma do algoritmo LRU simples.

Figura 8 – LRU-MIS execution flow



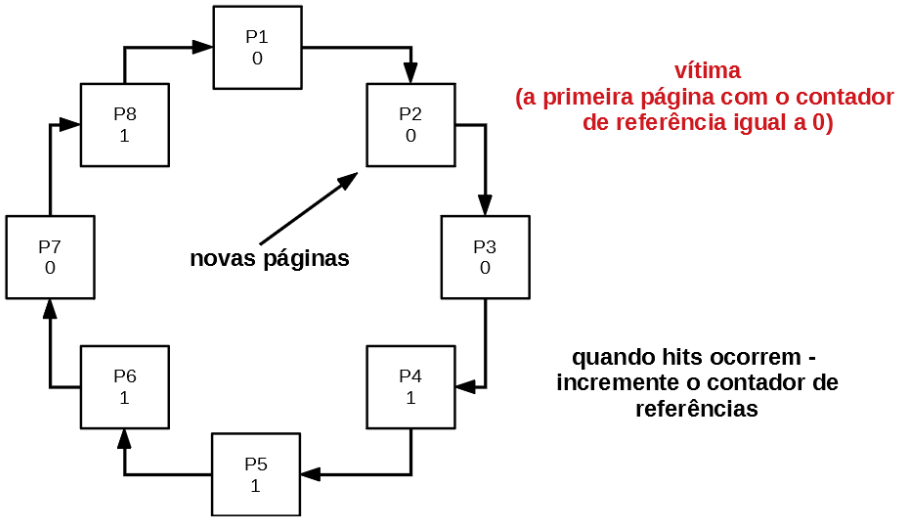

Fonte: O Autor.

A inserção de novas páginas no ponto médio evita que elas eventualmente poluam o *buffer* em caso de grandes varreduras sequenciais, preservando assim as páginas usadas mais recentemente que já foram referenciadas mais de uma vez. A localização do ponto médio é um parâmetro de ajuste. Os SGBDs comerciais ORACLE e MySQL usam estratégias semelhantes de inserção (ORACLE, 2021b) (ORACLE, 2021a). No caso do MySQL, é usado 62,5% (5/8) do *buffer* para páginas na região quente.

### *3.1.4 GCLOCK (Generalized Clock)*

*Clock*, também conhecido como *Second Chance*, é um algoritmo de substituição destinado a dar às páginas uma segunda chance no *buffer*. O nome *Clock* é devido à sua implementação usando uma lista circular de páginas com um ponteiro de página (CORBATÓ; TECHNOLOGY), 1968). GCLOCK é apenas uma generalização de *Clock*, dos quais os bits de referência não estão limitados a apenas 1 ou 0. GCLOCK usa um contador e dois parâmetros de ajuste: contador inicial *(IC)* e contador de *hits (HC)* (NICOLA *et al.*, 1992).

Figura 9 – GCLOCK fluxo de execução




Fonte: O Autor.

A Figura 9 descreve o fluxo de execução do GCLOCK. Cada página está associada a um contador de referência, semelhante à política LFU. Novas páginas são inseridas usando um ponteiro em uma lista circular com um contador de referência igual a *IC*. Quando ocorre um acerto, este contador é definido para o valor do *HC*. Quando o *buffer* está cheio, o ponteiro é consultado; se a contagem de referência for igual a 0, essa página é escolhida como vítima, a nova página é inserida nesta posição atual e o ponteiro avança para a próxima posição. Caso contrário, o algoritmo decrementa o contador de referência em 1 e o ponteiro avança para a próxima posição. Este processo é repetido até que seja encontrada uma página com o contador de referência igual a 0.

O SGBD comercial PostgreSQL implementa uma variante chamada *Clock Sweep* (XIA; BU, 2015) que considera tanto o contador de referência, mas também um sinalizador, que indica se uma transação estiver acessando esta página. Uma vantagem é que se a vítima for acessada por uma transação e não puder ser despejada, é bem simples escolher outra vítima avançando o ponteiro Clock. A desvantagem do Clock é que quando o ponteiro gira várias vezes procurando uma vítima pode dificultar a execução das transações.

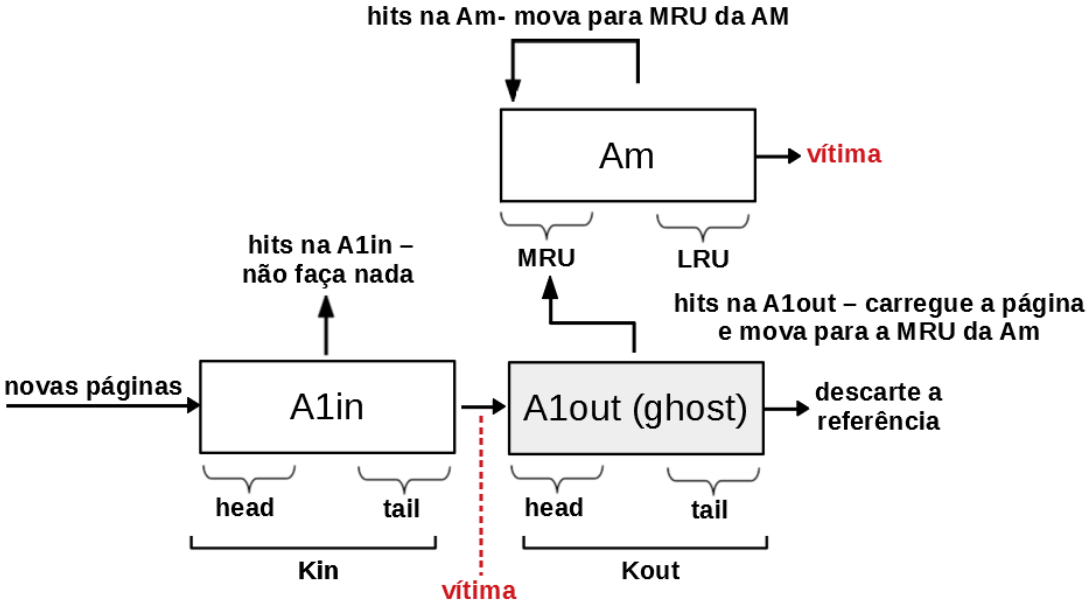
### *3.1.5 2Q*

Uma das motivações para a criação do algoritmo 2Q foi superar o algoritmo LRU-K, o problema com o LRU-K é que cada solicitação de página requer complexidade em O(log N), onde N representa o tamanho do *buffer*. Assim, o 2Q foi criado com o objetivo de ter um baixo

*overhead* com complexidade constante O(1) e desempenho em todas as requisições. O 2Q tem duas variações: a versão simplificada e a versão completa. Este trabalho descreverá a versão completa, que é composta por três filas, chamadas A1in, Am e A1ou (SHASHA; JOHNSON, 1994).

- A1in: Controla a recência das páginas usando o algoritmo FIFO.
- Am: Controla a frequência das páginas usando o algoritmo LRU.
- A1out: Recebe as páginas despejadas do A1in e determina quais páginas inserir no Am. A1out é uma lista *ghost*, pois os dados nas páginas não são mantidos na memória, apenas metadados, como os identificadores de páginas.

Figura 10 – 2Q fluxo de execução




Fonte: O Autor.

A Figura 10 descreve o fluxo de execução do 2Q. Em caso de *miss*, as novas páginas são carregadas na memória e inseridas na *head* de A1in. Quando ocorre *hits* em A1in nada é feito, porque as páginas com muitas referências próximas não são imediatamente consideradas páginas frequentes, as páginas são envelhecidas. *Hits* em Am significa que as páginas são movidas para o Am MRU. *Hits* em A1out significam que as páginas são carregadas na memória e inseridas no Am MRU.

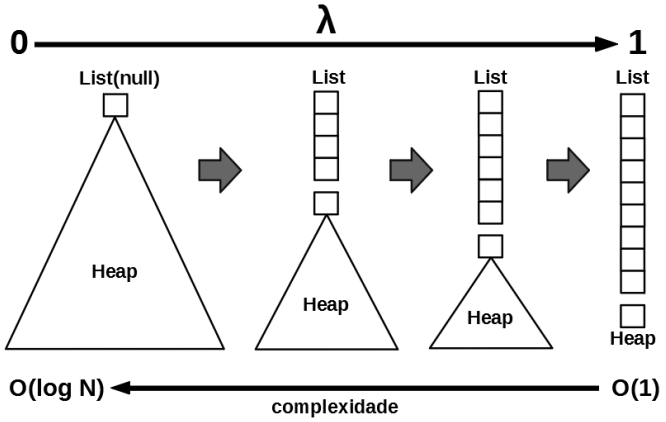
O espaço real do *buffer* é dividido entre A1in e Am (porque A1out é uma lista *ghost*). O *Kin* e *Kout* são parâmetros de ajuste que representam o tamanho máximo de crescimento de A1in e A1out, respectivamente. Quando o *buffer* está cheio e é necessária uma substituição, uma vítima é escolhida entre A1in ou Am, seguindo esta condição: se o tamanho atual de A1in

for maior que *Kin*, significa que A1in excedeu o tamanho máximo e a página final de A1in é despejada como vítima. Caso contrário, a página LRU de Am é escolhida como vítima. Todas as páginas despejadas do A1in são inseridas no A1out como *ghosts*. Se o tamanho atual de A1out for maior que *Kout*, a página do final de A1out *(tail)* será descartada.

### *3.1.6 LRFU*

Lidar com recência e frequência é um desafio na implementação de políticas de substituição. *Least Recently/Frequently Used* (LRFU) é uma política que assume um comportamento de escolha de vítimas entre LRU e LFU usando um parâmetro de ajuste $\lambda$. Em outras palavras, quando o valor de $\lambda$ é igual ou próximo de 0, o LRFU tende a funcionar de forma igual ou semelhante ao LFU. Quando o valor de $\lambda$ se aproxima de 1, a política LRFU tende a funcionar como uma política baseada em recência (KIM *et al.*, 1996).

Figura 11 – LRFU - impacto do $\lambda$ nas estruturas de dados




Fonte: O Autor.

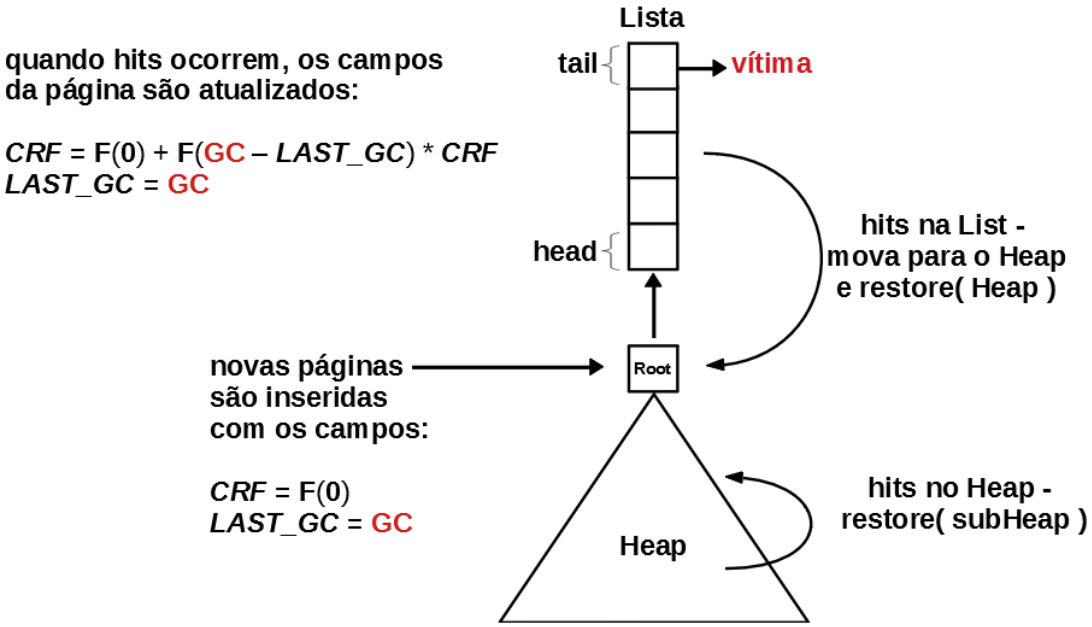
A LRFU pode ser implementada usando duas estruturas de dados, uma lista e um *heap*. O parâmetro $\lambda$ influencia diretamente no tamanho dessas estruturas, conforme mostrado na Figura 11. Assim, a complexidade computacional do algoritmo também depende do valor de $\lambda$, variando entre a complexidade das políticas LRU e LFU, O(1) e O(log N), respectivamente.

Na política LRFU, cada solicitação de página incrementa um contador global *(GC)*. Cada página tem dois campos. O primeiro é *LAST_GC*, é um campo que armazena o valor do contador global do momento em que a página foi referenciada pela última vez. O segundo campo, chamado *Combined Recency and Frequency* (CRF), determina a importância dessa página segundo o algoritmo. CRF é calculado com uma função de peso, usando a Fórmula 3.1, onde x é a distância de referências entre *GC* e *LAST_GC* daquela página específica.

$$F(x) = \frac{1}{2}^{\lambda x} \tag{3.1}$$

A Figura 12 descreve o fluxo de execução simplificado do LRFU. A estrutura do *heap* é organizada com o valor CRF de cada página. As novas páginas são inseridas na estrutura *heap* com as informações CRF e *LAST_GC*. Quando o *buffer* está cheio, a página antiga na raiz do *heap* é inserida no início da lista *(head)*. Em seguida, a página no final da lista *(tail)* é selecionada como vítima, pois o LRFU procura substituir as páginas com o menor CRF.

Figura 12 – LRFU fluxo de execução




Fonte: O Autor.

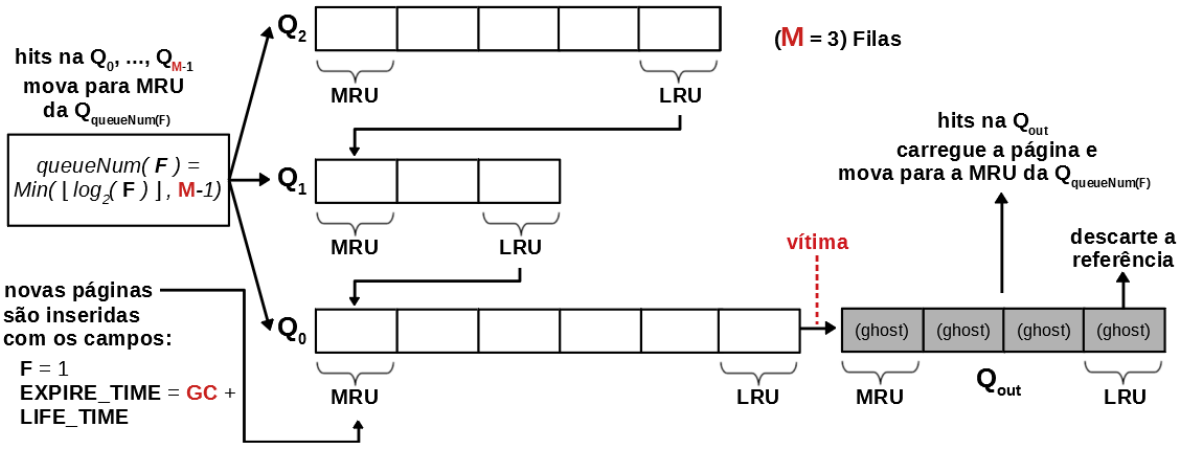
Sempre que ocorre um *hit*, os valores CRF e *LAST_GC* são atualizados. Se a página estiver na lista ela será promovida e inserida no *heap* e a estrutura será restaurada chamando a função *restore()* para o heap. Se a ocorrência ocorrer para uma página já dentro do *heap*, só será necessário restaurar uma parte do *heap (subheap)* referente essa página.

O LRFU possui uma variante chamada Window LRFU (BAI *et al.*, 2016), que busca se adaptar melhor as mudanças de padrões de acesso usando uma estratégia de janela deslizante que mantém um histórico das últimas requisições.

### *3.1.7 MQ*

O algoritmo *Multi-Queue* (MQ) apresenta um *buffer* com *M* filas LRU denominadas $Q_0$, ..., $Q_{M-1}$ com tamanhos dinâmicos ajustados durante a execução e uma fila *ghost* $Q_{out}$ com um tamanho fixo semelhante ao 2Q (ver Seção 3.1.5). O algoritmo prioriza páginas frequentes, mas ao mesmo tempo, aplica mecanismos de envelhecimento usando as filas como um ranking de páginas. *M*, $Q_{out}$ e *LIFE_TIME* são parâmetros de ajuste. *LIFE_TIME* indica o tempo de vida da página. O MQ mantém um contador global de referências *(GC)* e cada página possui um contador de referências *(F)*, e um campo chamado *EXPIRE_TIME* usado para envelhecimento, calculado pela soma do *GC* atual e *LIFE_TIME* (ZHOU *et al.*, 2001).

Figura 13 – MQ fluxo de execução




Fonte: O Autor.

A Figura 13 descreve o fluxo de execução do MQ. Novas páginas são inseridas no MRU de $Q_0$ com seu contador de frequência igual a 1 e o *EXPIRE_TIME* calculado. Quando ocorre um *hit* em uma página que reside em uma das filas *M*, o campo *F* é incrementado e *EXPIRE_TIME* é atualizado com o *GC* atual. A página é promovida para o lado MRU de uma das filas. O algoritmo usa a função, chamada *queueNum*, que indica em qual das filas *M* a página deve ser movida.

Quando o *buffer* está cheio, o MQ seleciona LRU da fila $Q_0$ como vítima. Se esta fila estiver vazia, a próxima LRU da fila superior será selecionada e assim por diante. A página vítima é movida para a fila $Q_{out}$ como uma *ghost*, preservando seus metadados, como o campo *F*. Se a fila $Q_{out}$ estiver cheia, a referência é finalmente descartada. Quando ocorre um *hit* na fila $Q_{out}$, a página é carregada no *buffer* e todos os campos são atualizados e a página é movida para uma das filas *M* usando a função *queueNum*.

No processo de envelhecimento, o algoritmo MQ verifica em cada solicitação as páginas LRU de cada uma das filas *M* (exceto $Q_0$). Se *EXPIRE_TIME* $<$ *GC*, a página é rebaixada para MRU da fila inferior, impedindo que esta página frequente resida no *buffer* por um período prolongado.

O MQ tem uma complexidade constante e usa um mecanismo de envelhecimento sem a necessidade de decrementar contadores. Além disso, quanto maior o número de filas, mais páginas podem ser analisadas pelo mecanismo de envelhecimento. No entanto, um número muito alto de filas pode afetar o tempo de execução e o consumo de memória.

### *3.1.8 LIRS*

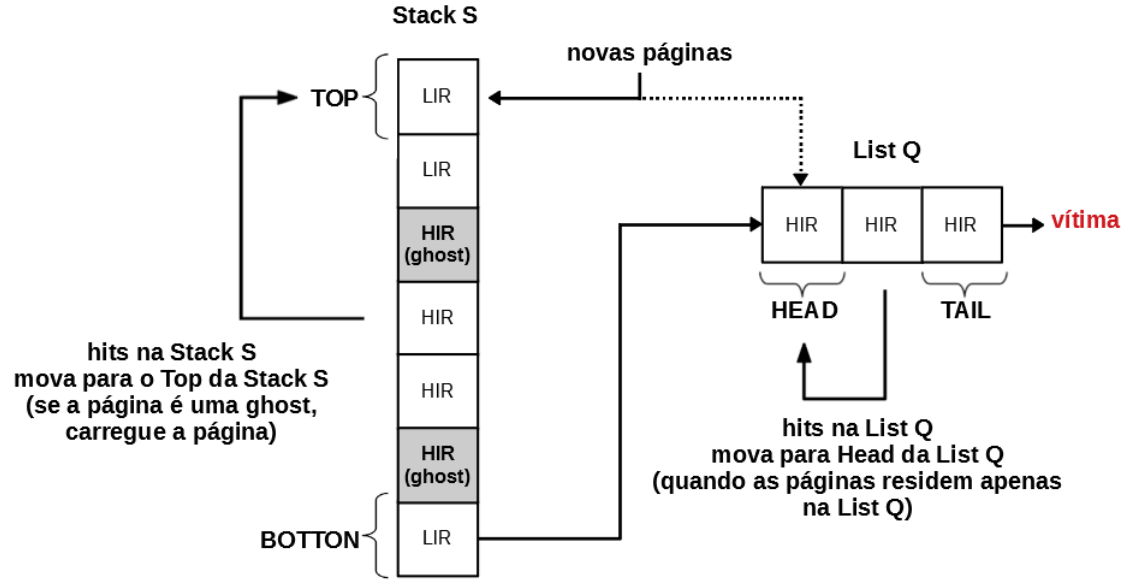
O *Low Inter-reference Recency Set* (LIRS) foi baseado no princípio *Inter-Reference Recency* (IRR). Considerando uma página P, o valor IRR(P) pode ser calculado como o número de acessos distintos de outras páginas entre dois acessos consecutivos de P, por exemplo, o último e o penúltimo acesso de P. Portanto, com este princípio, as páginas com um IRR baixo tendem a ser re-acessadas segundo o algoritmo (JIANG; ZHANG, 2002).

O LIRS classifica as páginas em dois tipos de status, LIR *(Low IRR)* e HIR *(High IRR)*. O objetivo é eliminar as páginas HIR do *buffer* primeiro. O tamanho do *buffer* em páginas é L, o algoritmo mantém um número máximo de páginas LIRs (LIR *size*) para as páginas classificadas como LIR e HIRs (HIR *size*) para as páginas classificadas como HIR, onde L = LIRs + HIRs. Esses dois tamanhos são parâmetros de ajuste. Os autores descrevem que o HIRs pode ter um tamanho pequeno, por exemplo, 1% de L.

A implementação do LIRS é baseada no algoritmo LRU e usa uma pilha S e uma lista Q. A Figura 14 descreve o fluxo de execução do LIRS. Novas páginas são inseridas no topo da pilha S com status LIR. Quando o LIRs (número de páginas LIR) é alcançado, as novas páginas são inseridas com status HIR no topo da pilha S e no topo da lista Q simultaneamente. Isso não significa que a página será duplicada porque as estruturas podem usar ponteiros para o mesmo elemento. Assim, uma página pode residir tanto na pilha S quanto na lista Q. No entanto, apenas páginas com status HIR podem residir na lista Q.

Quando novas páginas são inseridas e o *buffer* está cheio, os tamanhos LIRs e HIRs são alcançados e a estratégia escolhe a página final da lista Q *(tail)* como vítima. Se esta página também residir na pilha S, o algoritmo mantém apenas os metadados, convertendo-os em uma página HIR ghost. Existem quatro casos possíveis ao solicitar páginas que já residem no *buffer*

Figura 14 – LIRS fluxo de execução




Fonte: O Autor.

ou são páginas ghost:

- Caso 1: *Hits* em páginas LIR na pilha S, a página é movida para o topo da pilha S, de forma semelhante ao algoritmo LRU. Se esta página estiver na parte inferior da pilha S, a operação de remoção da pilha será executada. Essa operação garante que apenas as páginas com status LIR residam na parte inferior da pilha S. Todas as páginas com status HIR acima da parte inferior da pilha S são removidas até que uma nova página LIR seja encontrada.
- Caso 2: *Hits* em páginas HIR na pilha S, a página é movida para o topo da pilha S e é promovida ao status LIR, o algoritmo também remove sua cópia da lista Q. Além disso, é necessário rebaixar a página LIR da parte inferior da pilha S para o status HIR. Se a página rebaixada também residir na lista Q, o algoritmo também removerá essa cópia. Em seguida, a página rebaixada é inserida no *head* da lista Q e a operação de remoção de pilha é executada.
- Caso 3: *Hits* em páginas *ghost* HIR na pilha S são considerados uma falha *(miss)* porque os dados da página não residem no *buffer*. Antes que os dados da página possam ser recarregados, é necessário escolher uma vítima para liberar espaço. Então, o processo é semelhante ao Caso 2, mas sem remover uma cópia da lista Q.
- Caso 4: *Hits* na página HIR na lista Q e que não residem na pilha S. Uma nova entidade de página é inserida no topo da pilha S. O status HIR da página é mantido e a página é movida para *head* da lista Q.

Uma das limitações do algoritmo LIRS é a operação de remoção de pilha. Dependendo da carga de trabalho, a estratégia pode ter um grande número de páginas *ghost* HIR residindo na pilha S. Isso pode afetar a complexidade do algoritmo, que não será constante em cenários específicos.

CLOCK-Pro (JIANG *et al.*, 2005) combina o algoritmo *Clock* (ver Seção 3.1.4) com LIRS. O *Clock* agora tem páginas *ghost* e usa novos mecanismos para diferenciar entre páginas quentes e frias. Anos depois, o CLOCK-Pro+ (LI, 2019) combinou o antigo CLOCK-Pro com o algoritmo CAR (ver Seção 3.1.10) e propôs uma nova abordagem para se adaptar a diferentes padrões de cargas de trabalho.

Outra variante é o *Dynamic LIRS* (DLIRS) (LI, 2018), que combina os mecanismos de adaptação do ARC (ver Seção 3.1.9) com o LIRS. O DLIRS controla dinamicamente o número de páginas HIR ou LIR no *buffer*.

As variantes *Probability-based Adjustable algorithm on Low Inter-reference Recency Set* (PA-LIRS) (WANG *et al.*, 2020) e LIRS-WSR (JUNG *et al.*, 2007) combinam LIRS com mecanismos semelhantes a LRU-WSR (ver Seção 3.2.2), reduzindo o número de gravações e aproveitando a mídia assimétrica. A diferença é que o PA-LIRS usa mecanismos para ajustar o número de páginas HIR e LIR dinamicamente.

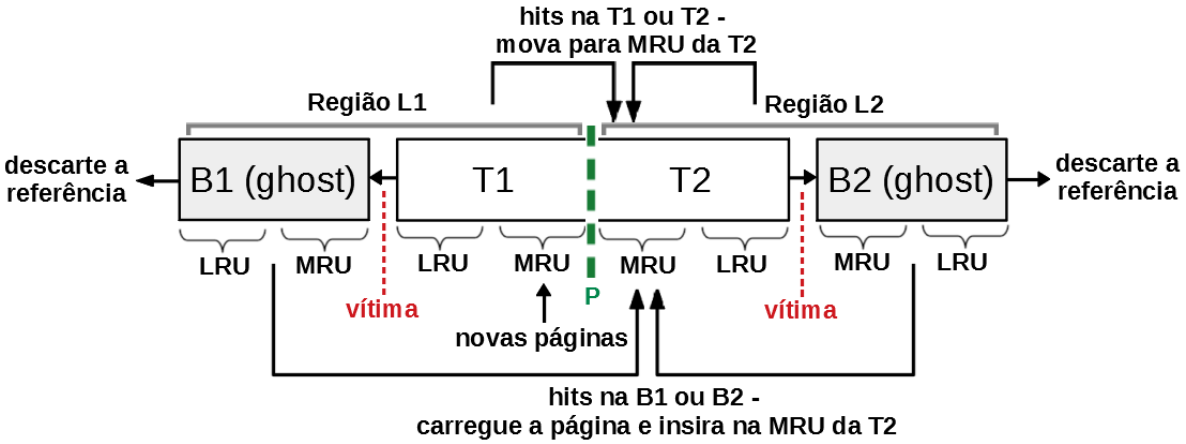
### *3.1.9 ARC*

O *Adaptive Replacement Cache* (ARC) foi projetado para dar suporte as mudanças de padrões das cargas de trabalho. O algoritmo equilibra a frequência e recência ao se adaptar dinamicamente e melhorar o *hit ratio* com uma complexidade constante. ARC é composto por duas regiões, L1 e L2, cada uma com duas LRUs. L1 é dividido em T1 e B1, e L2 é dividido em T2 e B2. A soma dos tamanhos de T1 e T2 deve ser $\leq$ C, onde C representa a capacidade total do *buffer* (MEGIDDO; MODHA, 2003).

- T1: Controla a recência das páginas usando o algoritmo LRU.
- T2: Controla a frequência das páginas usando o algoritmo LRU.
- B1: Recebe as páginas despejadas de T1 como *ghosts* e indica o crescimento de T1.
- B2: Recebe as páginas despejadas de T2 como *ghosts* e indica crescimento de T2.

A Figura 15 descreve o fluxo de execução do ARC. Em caso de falha *(miss)*, as novas páginas são carregadas na memória e inseridas na MRU de T1. Para *hits* em T1 ou T2 as páginas são movidas para MRU de T2. *Hits* em B1 ou B2, as páginas são carregadas na memória

Figura 15 – ARC fluxo de execução




Fonte: O Autor.

e inseridas no na MRU de T2. As vítimas de T1 e T2 vão para B1 e B2 como páginas *ghost*, respectivamente. As vítimas de B1 e B2 são descartadas.

P ($0 \leq P \leq C$) é uma variável que funciona como um *advisor*. Indicando o tamanho desejado para T1 e também é utilizado no momento da substituição. Inicialmente, o tamanho de P é igual a $|T1|$. *Hits* em B1 são indicativos de que T1 precisa de mais espaço, então P é incrementado seguindo a fórmula: $P = \min(C, P + \max( |B2| / |B1|, 1 ))$. Esta fórmula considera que se o tamanho de B2 for maior que B1, maior será o incremento de P. A mesma coisa acontece quando ARC tem *hits* em B2, mas usa a fórmula oposta para diminuir o tamanho: $P = \max(0, P - \max( |B1| / |B2|, 1) )$.

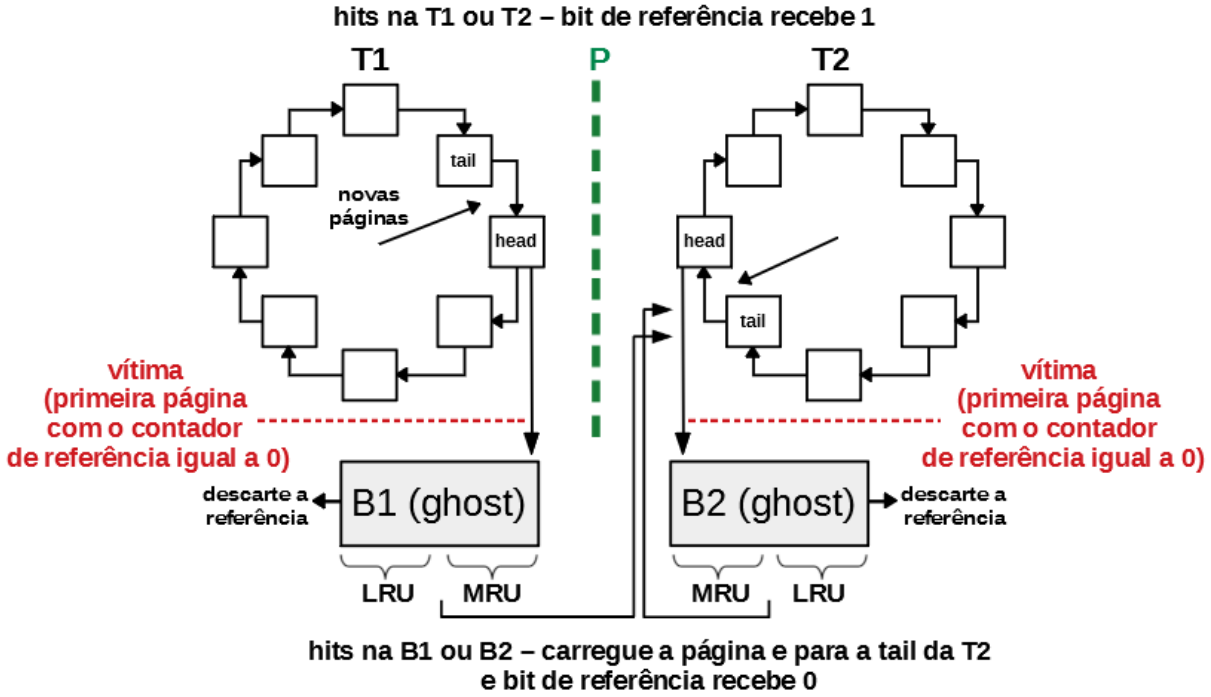
Quando $|T1| + |T2| = C$, o *buffer* está cheio, então o algoritmo deve escolher uma vítima de T1 ou T2 dependendo do valor de P. Grosso modo, se o tamanho atual de T1 for maior que P, o ARC deve escolher a LRU de T1 como vítima, caso contrário a LRU de T2. Em seguida, a vítima será movida para sua respectiva lista *ghost*. O ARC também possui um recurso *scan resistance* (ver Seção 2.3.1). B1 deve aumentar com recebimento de muitas páginas de T1 durante uma longa sequência de páginas recentes, poucos ou nenhum *hit* ocorrem em B1. Portanto, o algoritmo deve favorecer T2 mantendo páginas frequentes.

### *3.1.10 CAR*

*Clock with Adaptive Replacement* (CAR) combina as políticas de substituição ARC e *Clock* (ver Seções 3.1.9 e 3.1.4). Uma das vantagens de usar o *Clock* é que durante o processamento das transações, o algoritmo suporta uma melhor concorrência do que o algoritmo LRU. O CAR adapta vários mecanismos ARC, mas mantém o uso de T1 para recência e T2 para

frequência com dois *Clocks*. Cada página agora tem um bit de referência. As listas fantasmas B1 e B2 permanecem as mesmas do ARC (BANSAL; MODHA, 2004).

Figura 16 – CAR fluxo de execução




Fonte: O Autor.

A Figura 16 descreve o fluxo de execução do CAR. As novas páginas são inseridas na *tail* de T1 com o bit de referência igual a 0. Quando ocorrem *hits* em T1 ou T2, apenas o bit de referência é definido como 1. Quando há *hits* em B1 ou B2, o CAR inicialmente escolhe uma vítima, então a página é inserida na *tail* de T2 com o bit de referência igual a 0. O valor do *advisor* P é atualizado, semelhante ao ARC.

O CAR usa o *advisor* P para decidir uma vítima em T1 ou T2. Se a escolha for T1, o algoritmo seleciona a *head* de T1 como vítima se o bit de referência for igual a 0. Em seguida, a página é movida para a MRU de B1 como uma página *ghost*. Caso contrário, se o bit de referência for igual a 1, a página é movida para a *tail* de T2, e o bit de referência é definido como 0. O processo é repetido, verificando a nova *head* de T1. Se a escolha for T2, o procedimento é o mesmo, mas agora usando B2.

Em vários cenários de carga de trabalho, algumas páginas são referenciadas várias vezes em um curto período de tempo e podem não ser referenciadas no futuro. Esse comportamento faz com que o algoritmo classifique a página como frequente imediatamente. Os autores propõem uma variante chamada CART, que impõe testes mais rigorosos até que uma página seja

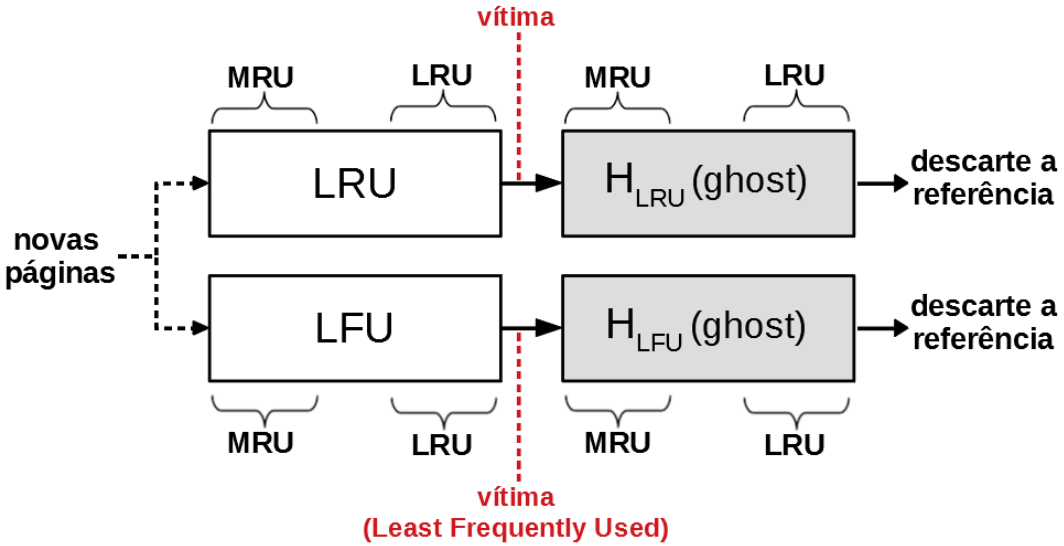
considerada frequente.

CAR e CART não possuem complexidade constante devido ao procedimento de escolha da vítima. No entanto, é possível aproveitar o mecanismo de escolha da vítima com os dois *Clocks* para melhorar a concorrência entre as transações.

### *3.1.11 LeCaR*

*Learning Cache Replacement* (LeCaR) é uma estratégia de substituição que usa técnicas de aprendizado de máquina. LeCaR usa o conceito de especialistas semelhante ao *Adaptive Caching Using Multiple Experts* (ACME) (ARI *et al.*, 2002). A ACME propõe um conjunto de políticas (especialistas) executadas simultaneamente a cada solicitação. A ACME calcula uma rank para decidir qual é a melhor estratégia no momento com base em suas estatísticas ao recompensar ou punir cada especialista. LeCaR usa *online reinforcement learning* com a técnica de *regret minimization*. Ao contrário do ACME, o LeCaR propõe uma abordagem mais minimalista usando dois especialistas, LRU e LFU, com o suporte de duas listas *ghost*, $H_{LRU}$ e $H_{LFU}$, e os pesos $W_{LRU}$ e $W_{LFU}$. Além disso, LeCaR usa dois parâmetros internos, a taxa de aprendizado e a taxa de desconto usada ao executar os mecanismos de aprendizado (VIETRI *et al.*, 2018).

Figura 17 – LeCaR fluxo de execução




Fonte: O Autor.

A Figura 17 descreve o fluxo de execução do LeCaR. Novas páginas são inseridas simultaneamente nas políticas LRU e LFU. Isso não significa que a página será duplicada. O algoritmo pode usar ponteiros nas entidades das estruturas que se referem à mesma página.

Quando ocorre um *hit*, ambas as estratégias atualizam sua estrutura, movendo a página para o lado MRU no caso do LRU e incrementando o contador de referência no caso do LFU. Quando ocorre uma ocorrência em qualquer lista *ghost*, os pesos $W_{LRU}$ ou $W_{LFU}$ são ajustados de acordo com a lista *ghost* que recebeu o *hit*.

O LeCaR primeiro seleciona um dos especialistas como vítima, com base em uma probabilidade aleatória e no valor dos pesos normalizados. Então, se o especialista LFU for escolhido, a página com o menor valor do contador de referência será removida de ambos os especialistas e enviada para a MRU da $H_{LFU}$ como uma *ghost*. Caso contrário, se o especialista LRU for escolhido, a página LRU será removida de ambos os especialistas e movida para $H_{LRU}$ como uma *ghost*.

O algoritmo LFU geralmente é implementado usando um *heap* (ver Seção 3.1.6). Portanto, a complexidade do LeCaR não é constante. O algoritmo CACHEUS (RODRIGUEZ *et al.*, 2021) aprimora o LeCaR usando especialistas mais sofisticados, oferecendo recursos como *scan resistance* (ver Seção 2.3.1) e *churn resistance*. O recurso *churn resistance* procura melhorar o desempenho dos *hit* quando o *buffer* lida com um conjunto de páginas maior que o tamanho do *buffer* que são acessados repetidamente com a mesma frequência.

### *3.1.12 Substituição baseada na abordagem Seq2Seq*

Hyejeong Choi e Sejin Park apresentam um algoritmo de substituição usando uma solução de aprendizado de máquina com foco na abordagem Seq2Seq baseada na rede neural LSTM (CHOI; PARK, 2022). Em resumo, o algoritmo faz predições das requisições de páginas futuras e seleciona como candidatas as vítimas que não constam na predição. Sua arquitetura é composta por seis componentes:

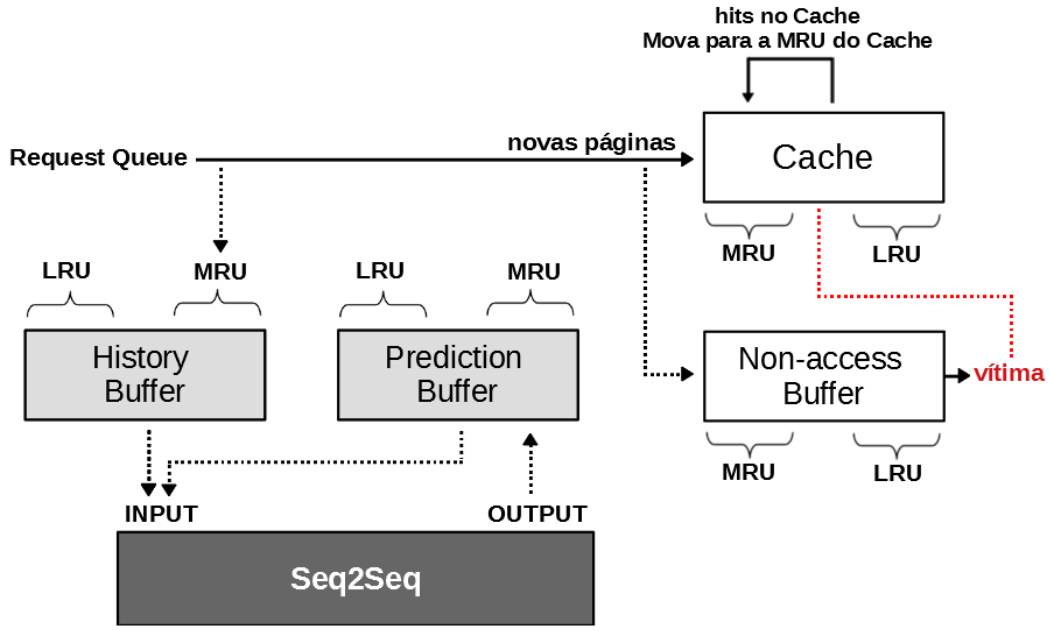
- *Cache*: Este componente é a base do algoritmo de substituição. Funciona de forma semelhante ao algoritmo LRU. Novas páginas são inseridas no lado da MRU e quando ocorre uma *hit*, a página é movida para o lado da MRU. No entanto, difere na seleção da vítima.
- Seq2Seq: Componente responsável por processar o modelo e fazer predições. A abordagem recebe como entrada um conjunto de informações de requisições de páginas de tamanho fixo. Este conjunto de entrada considera o identificador de página (número do bloco), frequência, *reuse distance* e delta, onde delta representa a diferença entre solicitações consecutivas. A abordagem então gera a previsão da próxima solicitação futura com o

possível identificador de página e o delta aproximado. baseada na abordagem Seq2Seq

- *Request Queue* e *History buffer*: A *Request Queue* captura os metadados da últimas requisições por página ao *buffer* na ordem temporal e usa essas informações para preencher o *History buffer* com um tamanho igual ao conjunto de entrada do Seq2Seq.
- *Prediction buffer*: Essa estrutura tem tamanho igual ao do Cache e é responsável por armazenar as informações de saída do Seq2Seq.
- Non-access *buffer*: Mantém as páginas que serão o alvo como candidatas à vítimas dependendo do processo de predição. O algoritmo julga que essas páginas não serão solicitadas no futuro.

O *Prediction buffer* e *History buffer* mantém apenas metadados de páginas que são gerados pelo Seq2Seq ou obtidos pela *Request Queue*. O componente Seq2Seq preenche o *Prediction buffer* inicialmente usando o *History buffer* como entrada. Porém, como a saída é limitada e são necessárias mais informações de entrada para o modelo, a estratégia opta por usar as informações de predições anteriores armazenadas no *Prediction buffer* como entrada para o Seq2Seq. Isso gera uma relação de dependência entre as predições no conjunto. Se uma única predições estiver errada, predições consecutivas também podem estar.

Figura 18 – Substituição baseada na abordagem Seq2Seq fluxo de execução




Fonte: O Autor.

A Figura 18 descreve fluxo de execução do algoritmo. Quando ocorre um *hit*, a página é movida para a MRU do *Cache*. O algoritmo faz predições para preencher o *Prediction*

*buffer* caso ocorra uma falha *(miss)* e o *buffer* esteja cheio. Em seguida, atualiza o *Non-access buffer* removendo as páginas presentes no *Prediction buffer*. Logo, de acordo com o algoritmo, o *Non-access buffer* reside as informações de páginas que não devem ser requisitas ao *buffer* em um futuro próximo. A página LRU do *Non-access buffer* é escolhida como vítima e é removida das estruturas *Cache* e *Non-access buffer*.

A abordagem também verifica se a predição está correta ao comparar novas requisições com o *Prediction buffer*. Quando o número de erros excede um determinado limite, o *Prediction buffer* é limpo e novas predições são feitas.

## 3.2 Políticas de substituição de *buffer* assimétricas

Esta seção foca nas políticas de substituição de *buffer* assimétricas. Com o emergência de novas tecnologias de armazenamento, surgiu uma nova classe de algoritmos de substituição. Essas estratégias agora consideram o tipo de operação (leitura/gravação) para aproveitar a assimetria da mídia de armazenamento, melhorando significativamente o desempenho. Algumas estratégias discutidas na Seção 3.1 também foram adaptadas para trabalhar com essa nova tecnologia de armazenamento.

Dividir o *buffer* em áreas de leitura e gravação é uma abordagem simples que oferece uma nova perspectiva no desenvolvimento de novos algoritmos. No entanto, apenas esta simples divisão não é suficiente para lidar com as questões de *hit ratio* e diferentes padrões de acesso. Outros aspectos, como a frequência de páginas, motivam combinações adicionais de algoritmos e estruturas para equilibrar o *hit ratio* e diminuir o número de gravações.

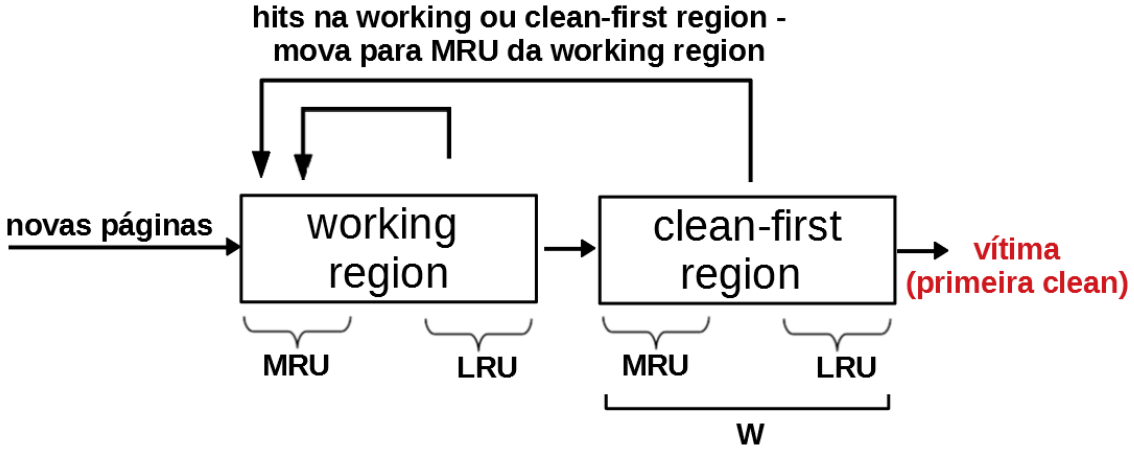
### *3.2.1 CFLRU*

*Clean-First LRU* (CFLRU) foi desenvolvido para trabalhar com mídia assimétrica. É a primeira estratégia endereçada à memória flash. O CFLRU tenta despejar primeiro as páginas leitura *(clean)*, evitando assim o custo das operações de escrita *(dirty)* durante as transações (PARK *et al.*, 2006).

A CFLRU divide a LRU em duas listas, chamadas de regiões *working* e *clean-first*. O parâmetro de ajuste *W* representa o tamanho *clean-first region* em páginas. O valor de *W* pode ser estático ou dinâmico. O método dinâmico calcula o valor de *W* coletando periodicamente estatísticas de operações de leitura e escrita e considera o custo dessas operações sobre a mídia

assimétrica.

Figura 19 – CFLRU fluxo de execução




Fonte: O Autor.

A Figura 19 descreve o fluxo de execução do CFLRU. Novas páginas são inseridas na MRU da *working region*. Se a lista estiver cheia, o algoritmo rebaixa a LRU da *working region* para a MRU da *clean-first region*. Em caso de *hit* em ambas as listas, a página é movida para a MRU da *working region*. Para escolher uma vítima, o algoritmo procura a página *clean* menos recente na *clean-first region*. Se nenhuma página *clean* for encontrada, a página LRU *dirty* da *clean-first region* será despejada. Portanto, o CFLRU não possui um tempo de execução constante, pois o algoritmo deve buscar $W$ páginas no pior caso, quando a *clean-first region* contiver apenas páginas *dirty*.

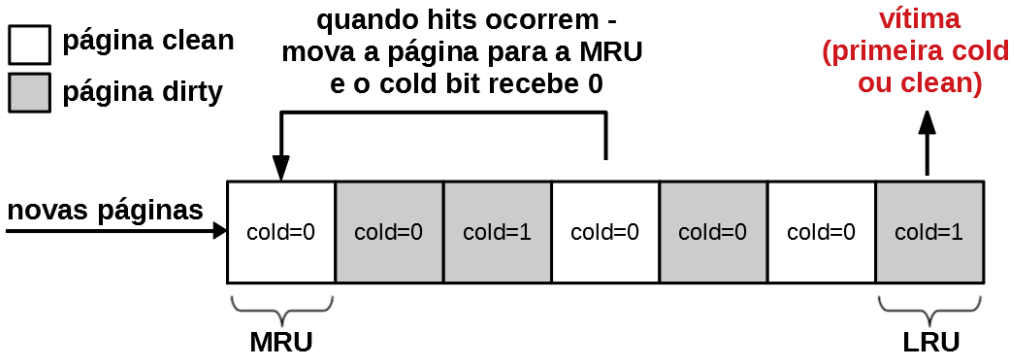
### *3.2.2 LRU-WSR*

*LRU Write Sequence Reordering* (LRU-WSR) é um algoritmo para mídia assimétrica que combina o algoritmo LRU e a heurística WSR. Esta heurística é baseada no algoritmo de segunda chance (ver Seção 3.1.4) (JUNG *et al.*, 2008).

A Figura 20 descreve o fluxo de execução do LRU-WSR. Novas páginas são inseridas na MRU com o *cold bit* igual a 0. Quando ocorre um *hit*, o *cold bit* é definido para 0 e a página é movida para o MRU. Quando o *buffer* está cheio e precisa escolher uma vítima, existem três casos possíveis:

- Caso 1: Se a página LRU é *clean*. A página é despejada independentemente do valor do *cold bit*.
- Caso 2: Se a página LRU é *dirty* com um *cold bit* igual a 0. O algoritmo deve dar uma

Figura 20 – LRU-WSR fluxo de execução




Fonte: O Autor.

segunda chance, esta página então é movida para o lado MRU, e o *cold bit* é definido como 1. O processo é então repetido para testar uma página na posição LRU.

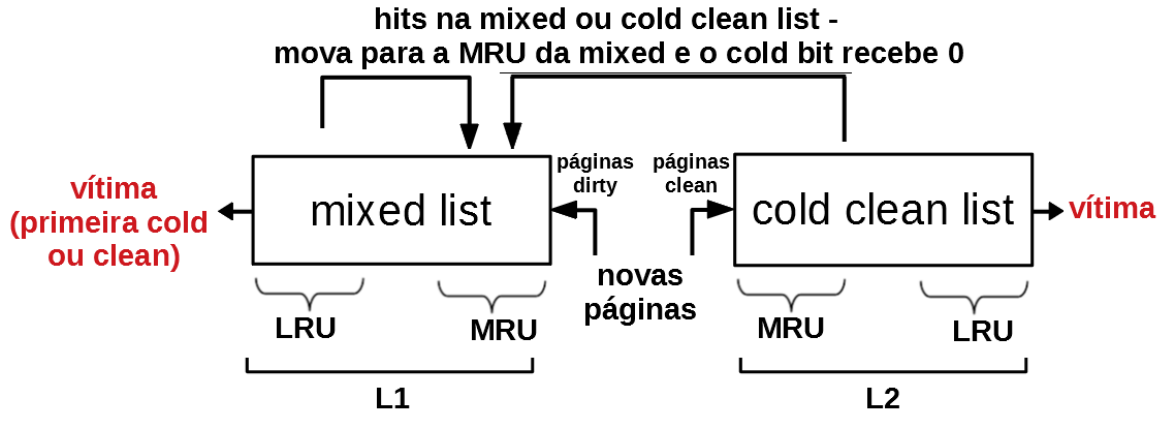
- Caso 3: Se a página LRU é *dirty* com um *cold bit* igual a 1. A página é despejada.

O LRU-WSR evita que páginas *dirty* permaneçam por muito tempo no *buffer* usando o status do *cold bit*. No entanto, sua desvantagem é que a complexidade não é constante, pois há casos em que é necessário mover várias páginas *dirty* para o lado da MRU até encontrar uma página *cold* ou *clean*.

### *3.2.3 CCF-LRU*

*Cold-Clean-First LRU* (CCF-LRU) é baseado nos algoritmos LRU-WSR e CFLRU (ver Seções 3.2.2 e 3.2.1). Essa abordagem dará às páginas *clean* uma segunda chance ao escolher as vítimas. O CCF-LRU contém duas listas, *mixed* e *cold-clean*, com tamanhos dinâmicos L1 e L2, respectivamente. Cada página também tem seu *cold bit* semelhante ao LRU-WSR (LI *et al.*, 2009).

Figura 21 – CCF-LRU fluxo de execução




Fonte: O Autor.

A Figura 21 descreve o fluxo de execução do CCF-LRU. Novas páginas são inseridas com um *cold bit* igual a 1 na MRU da lista *cold-clean*. O CCF-LRU foi proposto para sistemas baseados em memória flash. Inicialmente, as novas páginas são inseridas como páginas *clean* na lista *cold-clean*, então quando as páginas recebem uma atualização, elas são consideradas *dirty* e movidas para a lista *mixed*.

Quando ocorrem *hits* em ambas as listas, a página é movida para a MRU da lista *mixed* e o *cold bit* é definido para 0. Quando o *buffer* está cheio (isto é, Capacidade do *buffer* = L1 + L2), o algoritmo remove a LRU da lista *cold-clean*. Todavia, se esta lista estiver vazia, o algoritmo verifica o LRU da lista *mixed*, e há três casos possíveis:

- Caso 1: Se a página LRU tem um *cold bit* igual a 1. A página é imediatamente despejada independentemente de ser *dirty* ou *clean*.
- Caso 2: Se a página LRU é *clean* com um *cold bit* igual a 0. A página recebe uma segunda chance e é movida para a MRU da lista *cold-clean* com o *cold bit* igual a 1, e o procedimento é executado novamente para uma nova LRU da lista *mixed*.
- Caso 3: Se a página LRU é *dirty* com o *cold bit* igual a 1. A página recebe uma segunda chance e é movida para a MRU da lista *mixed* com o *cold bit* igual a 1, e o procedimento é executado novamente para uma nova LRU de a lista *mixed*.

Fica evidente que o CCF-LRU não possui complexidade constante, pois em cenários específicos quando a lista *cold-clean* está vazia, é necessário encontrar uma página *cold* na lista *mixed*. Apesar disso, há outro problema de *cache starvation* (ver Seção 2.3.2). Quando a lista *cold-clean* estiver vazia, as novas páginas inseridas na *cold-clean* sempre serão indicadas como vítimas imediatamente, evitando que essa lista cresça e assim podendo diminuir o *hit ratio*.

*Controllable Cold Clean First Least Recently Used* (CCCF-LRU) é uma melhoria do CCF-LRU. A diferença é que faz uso do parâmetro *minCCL* que determina o tamanho mínimo da lista *cold-clean*, evitando que a lista fique muito pequena. Se este tamanho mínimo for alcançado, o algoritmo procura uma vítima na lista *mixed* (XIA; BU, 2015).

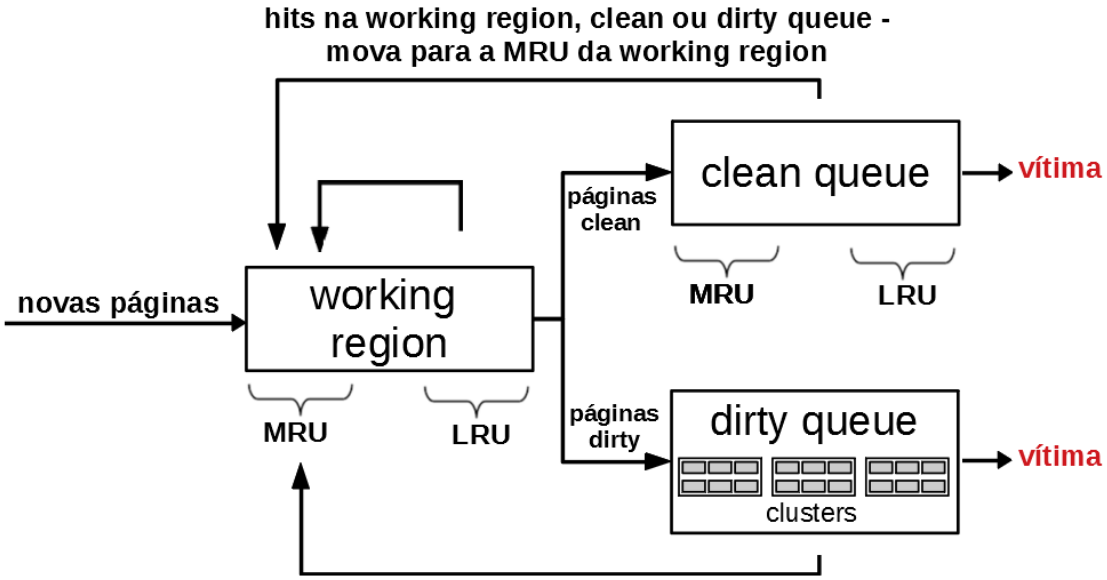
### *3.2.4 CFDC*

*Clean-First Dirty-Clustered* (CFDC) é um algoritmo baseado no CFLRU (ver Seção 3.2.1) e explora melhorias no desempenho da memória flash usando técnicas baseadas em blocos semelhantes aos algoritmos *Block-Level LRU* (BPLRU) (KIM; AHN, 2008) e *Flash-Aware buffer* (FAB) (JO *et al.*, 2006). Em resumo, esses algoritmos usam estratégias para agrupar páginas,

melhorando assim o desempenho da memória flash (OU *et al.*, 2009).

O CFDC é composto pela *working region* e *priority region*. A *working region* pode ser implementada usando outro algoritmo de *buffer*, por exemplo, LRU. A *priority region* é composta pelas filas *clean* e *dirty*. O parâmetro de ajuste *priority window* determina o tamanho da região de prioridade. Durante os testes, os autores usam 50% do tamanho total do *buffer* para cada região. Os tamanhos de fila *clean* e *dirty* são dinâmicos, mas a soma das filas não excede o valor da *priority window*.

Figura 22 – CFDC fluxo de execução




Fonte: O Autor.

A Figura 22 descreve o fluxo de execução do CFDC. Novas páginas são inseridas na *working region*. Quando a *working region* está cheia, a sua página LRU é movida para filas *clean* ou *dirty* de acordo com seu tipo de operação. A fila *dirty* é uma fila de prioridade de *clusters*. O CFDC usa uma estrutura de *hash* para mapear em qual *cluster* uma página deve residir. O algoritmo também calcula uma prioridade para cada *cluster* que determina a importância do *cluster* para o algoritmo.

Quando ocorre *hits* em qualquer uma das estruturas, as páginas são movidas para a MRU da *working region*. Quando é necessário escolher uma vítima, se a fila *clean* não estiver vazia, o algoritmo seleciona a LRU como vítima. Caso contrário, o algoritmo despeja a primeira página do *cluster* com a prioridade mais baixa da fila *dirty*. Em seguida, esse *cluster* é marcado para que suas páginas sejam as próximas vítimas até que o *cluster* esteja vazio. Em vista disso, há uma alta probabilidade de que as páginas no mesmo *cluster* sejam liberadas para o mesmo

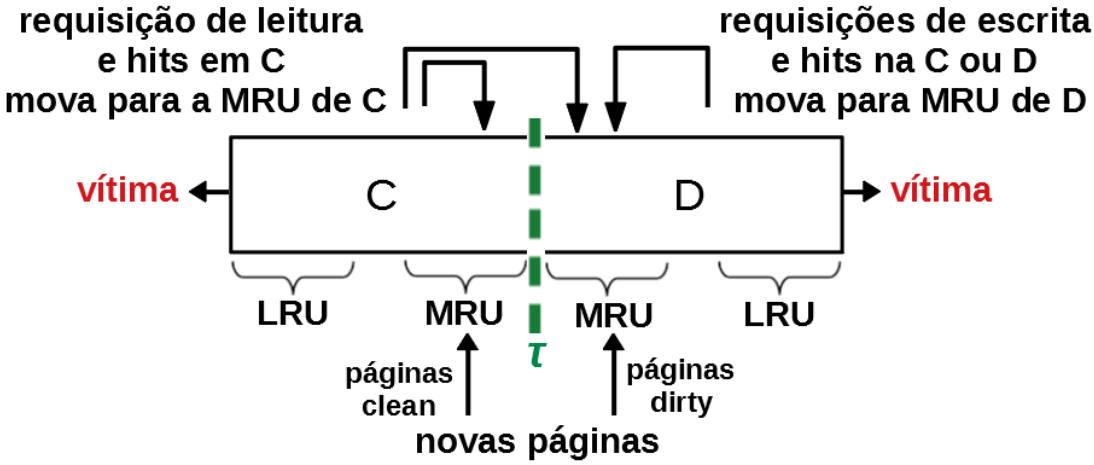
bloco de memória flash por conta da proximidade dos seus identificadores dentro do *cluster*.

A complexidade do CFDC não é constante devido aos mecanismos de agrupamento. No entanto, ele também apresenta o problema de cache starvation (ver Seção 2.3.2) semelhante ao algoritmo CCF-LRU (consulte a Seção 3.2.3). Quando a fila *clean* estiver vazia e uma nova página *clean* for inserida, a próxima vítima será imediatamente aquela página recente, evitando que a fila *clean* cresça podendo diminuir o desempenho no *hit ratio*.

### *3.2.5 CASA*

*Cost-Aware Self-Adaptive* (CASA) é uma política de substituição de páginas para mídia assimétrica. O CASA divide o *buffer* em duas regiões com tamanhos dinâmicos, |C| e |D|, que são listas *clean* e *dirty*, respectivamente. Em que |C| + |D| = B, sendo B o tamanho do *buffer* em páginas. O algoritmo usa o *advisor* $\tau$ para controlar o tamanho das listas. O CASA usa dois parâmetros de ajuste, $C_R$ e $C_W$, que representam o custo das operações de leitura ou escrita e são usados quando o *advisor* $\tau$ é ajustado (OU; HäRDER, 2010).

Figura 23 – CASA fluxo de execução




Fonte: O Autor.

A Figura 23 descreve o fluxo de execução do CASA. Novas páginas são inseridas na MRU de C ou D, dependendo do tipo de operação. Para selecionar uma vítima, o CASA verifica o *advisor* $\tau$. Se $|C| > \tau$, então a vítima será a LRU de C. Caso contrário, a LRU de D. Quando ocorre um *hit*, o algoritmo tem quatro casos:

- Caso 1: *Hits* em C com a operação de leitura. A página é movida para o MRU de C e o *advisor* $\tau$ é ajustado para $\tau = \min(\tau + C_R \times (|D| / |C|), B)$.
- Caso 2: *Hits* em D com a operação de escrita. A página é movida para o MRU de D e o

*advisor* $\tau$ é ajustado para $\tau = \max(\tau - C_W \times (|C| / |D|), 0)$.

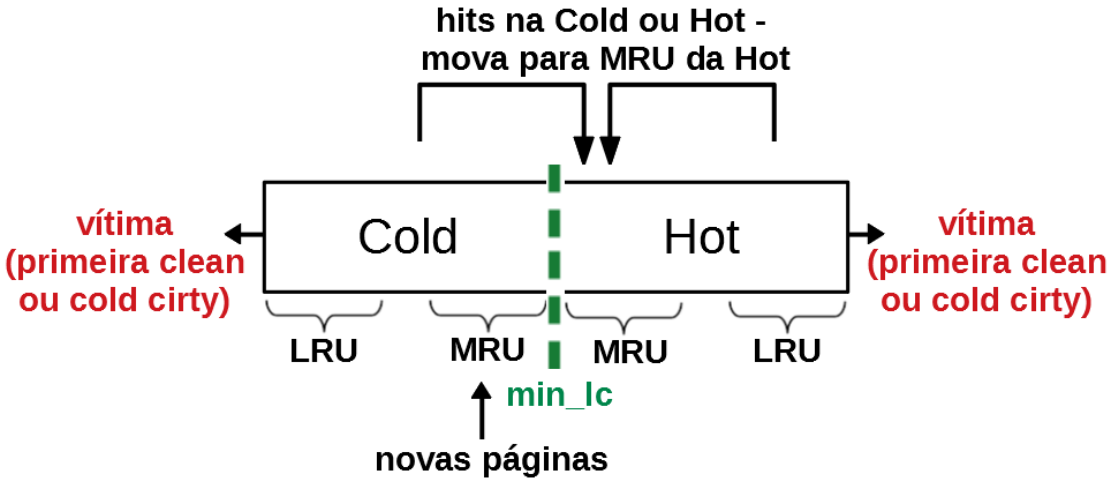
- Caso 3: *Hits* em C com a operação de escrita. A página é movida para o MRU de D.
- Caso 4: *Hits* em D com a operação de leitura. A página é movida para o MRU de D.

Existem outros algoritmos semelhantes ao CASA. Por exemplo, FD-Buffer (ON *et al.*, 2010) que também usa duas listas LRU para as páginas de leitura e escrita, mas cujo algoritmo pode ser configurado para que a lista de escrita seja substituída por uma estratégia de agrupamento, semelhante ao CFDC. Anos depois, o *Self Adaptive with Write Clustering* (SAWC) (OU *et al.*, 2013) melhorou o CASA usando também uma estratégia de *clustering*. O HDC (LIN *et al.*, 2014) também usa duas regiões leitura e escrita, mas seleciona uma vítima usando critérios como o *hot degree* da página e o custo da escrita em mídia assimétrica.

### *3.2.6 AD-LRU*

*Adaptive Double LRU* (AD-LRU) é uma estratégia para mídia assimétrica que remove as páginas *clean* primeiro e dá uma segunda chance às páginas *dirty*. O AD-LRU divide o *buffer* em duas filas: *cold* e *hot*. O tamanho das duas filas é ajustado dinamicamente, e o parâmetro de ajuste *min_lc* indica o tamanho mínimo da fila *cold*. Cada fila tem um ponteiro chamado FC que faz referência à página *clean* menos recentemente usada. Os ponteiros FC são ajustados durante cada solicitação (JIN *et al.*, 2012).

Figura 24 – AD-LRU fluxo de execução




Fonte: O Autor.

A Figura 24 descreve o fluxo de execução do AD-LRU. Cada página contém um *cold bit* semelhante ao LRU-WSR (ver Seção 3.2.2). Novas páginas são inseridas na MRU da fila *cold* com o *cold bit* igual a 0 e o ponteiro FC da fila *cold* é ajustado. Quando ocorre um *hit*, a

página é movida para a MRU da fila *hot* e o *cold bit* é definido como 0. Se o *hit* foi na fila *hot*, somente o FC da fila *hot* é ajustado. Caso contrário, o AD-LRU ajusta o FC das duas filas.

Para substituir uma página o AD-LRU primeiro seleciona uma das duas filas usando o parâmetro *min_lc* quando o *buffer* está cheio. A fila *cold* é selecionada como vítima se $C >$ *min_lc*, em que *C* é o número de páginas da fila *cold*. Caso contrário, o algoritmo seleciona a fila *hot*. Em seguida, se o ponteiro FC da fila escolhida não for nulo, o algoritmo seleciona a página referenciada pelo ponteiro FC como vítima e depois ajusta o ponteiro FC para referenciar uma nova página *clean* desta fila caso exista.

Se FC for nulo, o algoritmo deve remover a página LRU *dirty*. No entanto, se a página tem um *cold bit* igual a 1, semelhante ao LRU-WSR, o algoritmo dá uma segunda chance, e define o *cold bit* como 0 e move a página para o lado MRU da fila selecionada, então o processo é repetido. Se o *cold bit* for igual a 0, a página LRU *dirty* será finalmente despejada.

O parâmetro *min_lc* controla o tamanho da fila *cold*, evitando que ela fique muito pequena ou fique muito grande. Consequentemente, *min_lc* evita o *cache starvation* e fornece uma *scan resistance* (ver Seções 2.3.2 e 2.3.1). O AD-LRU não possui complexidade constante devido à heurística da segunda chance, semelhante ao LRU-WSR.

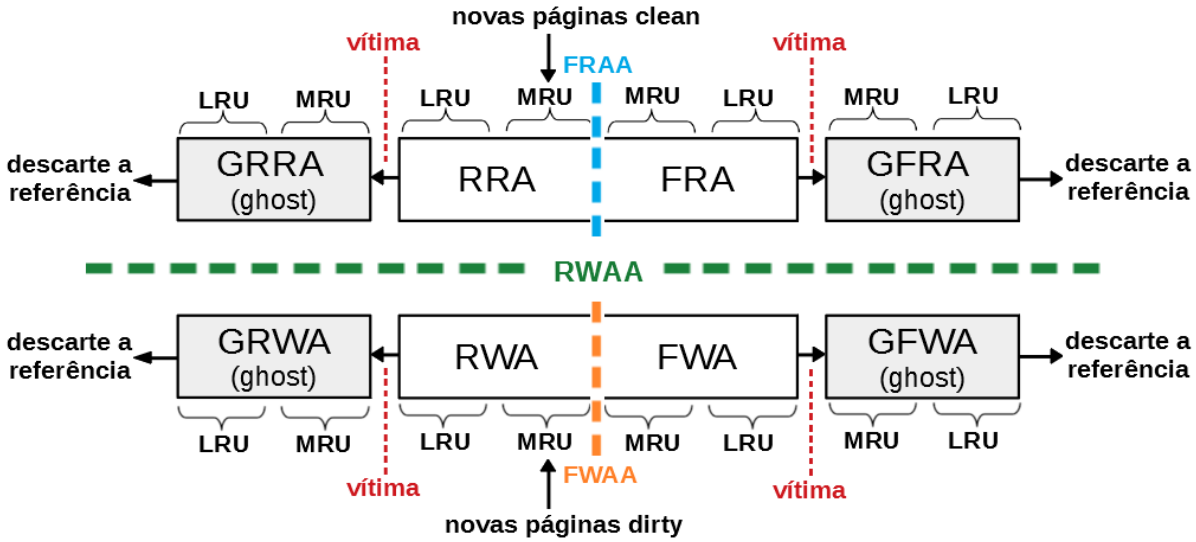
### *3.2.7 SCMBP-SCCW*

SCMBP-SCCW é uma política de substituição para mídia assimétrica. O algoritmo é baseado em *Hierarchical Adaptive Replacement Cache* (H-ARC) (FAN *et al.*, 2014) e no ARC (ver Seção 3.1.9) com características de CFDC (ver Seção 3.2.4). O SCMBP-SCCW utiliza os benefícios dos mecanismos de adaptação do ARC (recência e frequência) aplicados para se adaptar às mudanças na carga de trabalho como a leitura e escrita de páginas. Ele também usa o mecanismo de agrupamento CFDC na sua *Recency Write Area* (RWA) (TAVARES, 2015).

A ideia é usar dois ARCs chamados RR *(Read Region)* e WR *(Write Region)*. O *buffer* é dividido em quatro regiões com quatro listas *ghosts* auxiliares. O *advisor* externo RWAA *(Read/Write Area Advisor)* controla os tamanhos de RR e WR. Os consultores internos FRAA *(Frequency Read Area Advisor)* e FWAA *(Frequency Write Area Advisor)* controlam os tamanhos das regiões de recência e frequência.

A Figura 25 descreve o fluxo de execução do SCMBP-SCCW. Novas páginas são inseridas na MRU da RRA *(Recent Read Area)* ou na RWA *(Recency Write Area)* dependendo do tipo de operação (leitura/gravação) e o *advisor* RWAA é ajustado. Quando ocorre um *hit*, a

Figura 25 – SCMBP-SCCW fluxo de execução




Fonte: O Autor.

página é movida para FRA *(Frequency Read Area)* ou FWA *(Frequency Write Area)*, dependendo do tipo de operação. Se o *hit* foi nas listas *ghosts*, a página é carregada e o *advisor* RWAA é ajustado, junto com os *advisors* FRAA ou FWAA, dependendo da operação.

Na seleção da vítima, o algoritmo primeiro escolhe entre RR ou WR usando o *advisor* RWAA. Em seguida, os *advisors* internos FRAA ou FWAA decidem entre recência ou frequência despejando a página LRU. No entanto, se a vítima residir no RWA, as páginas do *cluster* com menor prioridade são selecionadas como vítimas durante as solicitações subsequentes, semelhante ao CFDC.

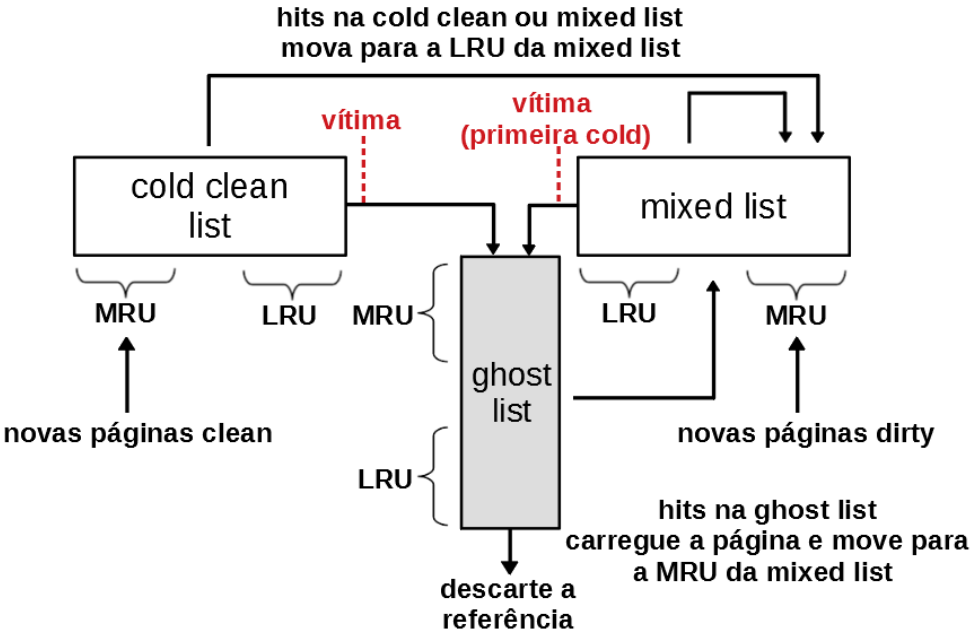
Ao atualizar os *advisors*, o algoritmo usa vários parâmetros de ajuste que ajudam a regular os tamanhos das regiões. Portanto, é possível impulsionar as regiões de escrita a crescerem mais rapidamente do que as de leitura, reduzindo o número de escritas e favorecendo mídias assimétricas. Da mesma forma que o ARC, o SCMBP-SCCW possui mecanismos de *scan resistance* (ver Seção 2.3.1), mas não possui uma complexidade constante devido aos mecanismos de agrupamento.

### *3.2.8 GASA*

*Ghost buffer Assisted and Self-tuning Algorithm* (GASA) é uma estratégia na qual o *buffer* é dividido em duas listas, *cold clean* e *mixed*, e usa uma lista *ghost* auxiliar. Os tamanhos das três listas são ajustados dinamicamente e cada página tem uma *hot-flag* e uma *ghost-flag*. A *hot-flag* é usada para dar uma segunda chance às páginas *hot*, segundo o algoritmo. A *ghost-flag*

é usado para ajustar o tamanho da lista *ghost* GS, que usa os parâmetros de ajuste *GSMIN* e *GSMAX*, indicando o tamanho mínimo e máximo (LI *et al.*, 2016).

Figura 26 – GASA fluxo de execução




Fonte: O Autor.

A Figura 26 descreve o fluxo de execução do GASA. Novas páginas são inseridas na MRU da lista *cold clean* ou *mixed* dependendo do tipo de operação com *hot-flag* e *ghost-flag* igual a 0. Para *hits* na lista *cold clean* ou *mixed*, a página é movida para a MRU da lista *mixed* e a *hot-flag* é definida como 1. Porém, se a página tiver uma *ghost-flag* igual a 1, o tamanho da lista *ghost* é incrementado GS = min(GS + 1, GSMAX) e a *ghost-flag* é definida como 0. Para *hits* na lista *ghost*, a página é carregada na memória e movida para o MRU da lista *mixed* e a *ghost-flag* e *hot-flag* são definidas como 1.

Quando o *buffer* está cheio e uma vítima precisa ser escolhida, se a lista *cold clean* não estiver vazia, o algoritmo seleciona a sua LRU como vítima. Caso contrário, o algoritmo seleciona o LRU da lista *mixed* como vítima se o *hot-flag* for igual a 0. Se o *hot-flag* for igual a 1, o GASA lhe dará uma segunda chance, atualizando a *hot-flag* para 0 e movendo a página para uma lista *cold clean* ou *mixed*, dependendo do tipo de operação. O processo é repetido para a nova LRU da lista *cold clean* ou *mixed*. Após selecionar uma vítima, seus metadados são inseridos no MRU da lista *ghost*. No entanto, se a vítima tiver a *ghost-flag* igual a 1, o tamanho da lista *ghost* é decrementado GS = max(GSMIN, GS – GSMAX/(GSMAX-GS+1)).

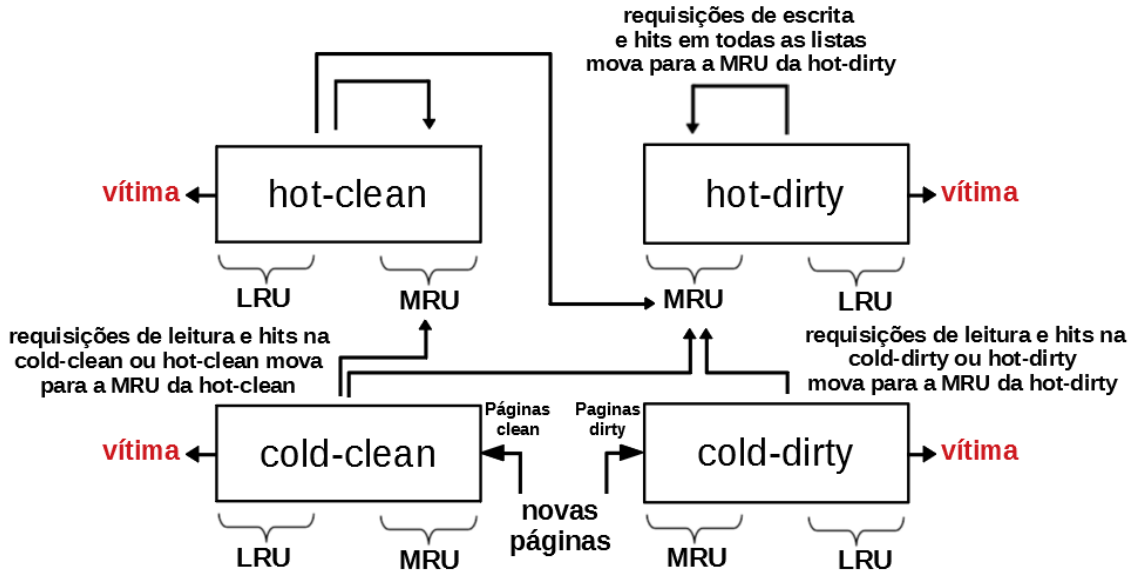
O GASA não tem uma complexidade constante devido ao processo de escolha da

vítima. O algoritmo propõe ajustar o tamanho da lista de *ghost*. Se o tamanho for pequeno, pode dificultar a identificação das páginas *hot*. Se o tamanho for grande, as páginas da lista *cold clean* terão uma alta probabilidade de serem identificadas como *hot* segundo o algoritmo, o que pode aumentar o número de gravações.

### *3.2.9 LLRU*

*Locality-aware Least Recently Used* (LLRU) é uma política de substituição que divide o *buffer* em quatro listas LRU: *cold-clean*, *cold-dirty*, *hot-clean* e *hot-dirty*. O tamanho de cada lista é dinâmico e cada página possui um contador de referência. O algoritmo usa dois parâmetros de ajuste: $E_{CC}$ e $E_{CD}$, que correspondem ao custo de leitura e gravação respectivamente (HE *et al.*, 2017).

Figura 27 – LLRU fluxo de execução




Fonte: O Autor.

A Figura 27 descreve o fluxo de execução da LLRU. Novas páginas são inseridas na *cold-clean* ou *cold-dirty*, dependendo do tipo de operação. Quando ocorre um *hit*, a página é movida para a MRU da *hot-dirty* se a operação for de escrita em qualquer uma das quatro listas. Se a operação for leitura e ocorre um *hit* na lista *cold-clean* ou *hot-clean*, a página será movida para a MRU da *hot-clean*. Por fim, se a operação for leitura e ocorrer um *hit* na *cold-dirty* ou *hot-dirty*, a página será movida para o MRU da *hot-dirty*.

O algoritmo primeiro calcula o custo da página *(PC)* para cada LRU das quatro listas para selecionar uma vítima. *PC* é calculado pela fórmula 3.2, em que *AT* representa o número de

acessos dessa página. Entre as quatro páginas LRU em cada lista, a vítima será a página com o menor *PC*.

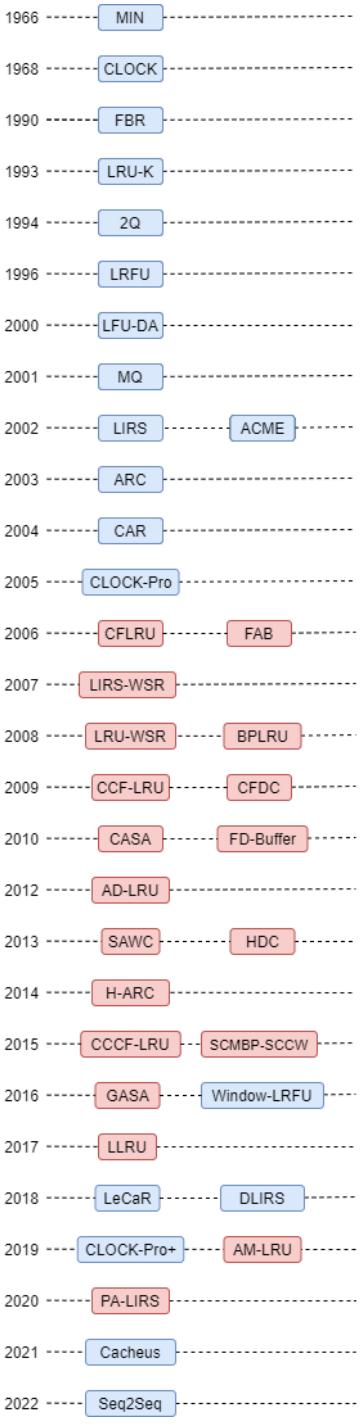
$$PC(p) = \begin{cases} AT(p) * E_{\mathrm{CC}}, & \text{if } p \text{ is } clean. \\ AT(p) * E_{\mathrm{CD}}, & \text{if } p \text{ is } dirty. \end{cases} \tag{3.2}$$

O LLRU tem complexidade constante e uma das suas desvantagens é que pode sofrer com o problema de *cache starvation* (ver Seção 2.3.2). As páginas da lista *cold-clean* têm alta probabilidade de sempre serem selecionadas como vítimas, impedindo que a lista cresça. O AM-LRU (WU *et al.*, 2019) foi proposto para melhorar o algoritmo LLRU usando um parâmetro de ajuste chamado *min_len*, semelhante ao AD-LRU (ver Seção 3.2.6). O *min_len* controla o tamanho da *cold-clean*, evitando que a lista fique muito pequena.

## 3.3 Linha do tempo histórica

A Figura 28 mostra uma linha do tempo com alguns algoritmos apresentados neste trabalho. Os algoritmos em azul foram desenvolvidos inicialmente para mídias não assimétricas; os algoritmos em vermelho são para mídia assimétrica. Em 1966, L. A. Belady publicou seus estudos sobre algoritmos de substituição, onde introduziu o algoritmo MIN. Nos anos seguintes, o algoritmo MIN tornou-se um recurso essencial para avaliar o *hit ratio* de novos algoritmos. Em 1968, o algoritmo *Clock* foi proposto e influenciou diversas estratégias nos anos futuros, apresentando a ideia de uma segunda chance para as páginas.

Figura 28 – Linha do tempo das políticas de *buffer*




Fonte: O Autor.

Nos anos seguintes, vários algoritmos foram propostos para diferenciar bem as páginas frequentes e lidar com a recência. Algoritmos como LRU-K e LFU-DA usam estratégias para minimizar que páginas muito frequentes residam por muito tempo no *buffer* sem receberem *hits*. Algoritmos como FBR e LRFU exploram o uso dos algoritmos LRU e LFU como base para lidar com recência e frequência.

Em 1994, o algoritmo 2Q optou por uma abordagem mais simplificada utilizando apenas listas com complexidade constante e foi provavelmente um dos primeiros a utilizar o conceito de listas *ghost* (ver Seção 2.3.3). O algoritmo 2Q também pode ter influenciado a criação de outros algoritmos, como MQ e ARC. O algoritmo LIRS apresenta o método de classificação das páginas como LIR e HIR (ver Seção 3.1.8). O algoritmo ARC inova com o conceito de adaptação utilizando o *advisor* que controla dinamicamente o tamanho das listas. LIRS e ARC são políticas importantes que influenciaram diversas variantes ou mecanismos semelhantes para melhorar essas estratégias.

Em 2006, tivemos um marco histórico. Devido à popularidade dos SSDs com seus

benefícios, não demorou muito para que surgissem novas estratégias que consideram a assimetria das mídias de armazenamento, como CFLRU e FAB. O CFLRU teve grande destaque nos anos seguintes. Alguns de seus problemas foram explorados e resolvidos por outros algoritmos. O LRU-WSR possivelmente também influenciou diversos outros algoritmos devido à estratégia de segunda chance para páginas *dirty*.

Algoritmos como CFDC, entre outros, tentaram extrair ainda mais desempenho do hardware com suas estratégias de *clustering*. H-ARC e SCMBP-SCCW atualizaram o algoritmo ARC para trabalhar com mídia assimétrica. Da mesma forma, LIRS-WSR e PA-LIRS também atualizaram o LIRS. Um dos desafios mais significativos nos dias de hoje é construir algoritmos que mantenham um equilíbrio entre recência, frequência, leitura e escrita. O atraso na remoção de páginas *dirty* é essencial devido à assimetria da mídia de armazenamento. No entanto, se páginas *clean* são frequentemente candidatas a vítimas e são imediatamente substituídas, tendo um curto tempo de vida pode comprometer diretamente o *hit ratio*. O algoritmo deve se adaptar a essas mudanças de carga de trabalho e manter um equilíbrio.

Abordagens como CFLRU, CCF-LRU, CFDC, AD-LRU, GASA e CCCF-LRU geralmente dividem o *buffer* em uma região fria *(cold)* e uma região quente *(hot)*. Em algumas dessas abordagens, na região fria reside apenas páginas *clean*. Acessos em páginas *clean* ou *dirty* promovem as páginas para a região quente. A preferência é despejar páginas de *cold-clean* ou páginas menos recentes sempre que possível.

Abordagens como CASA, FD-Buffer, SAWC e HDC dividem o *buffer* em regiões de leitura e escrita, geralmente usando um *advisor* para controlar o tamanho das regiões e indicar qual região será a vítima.

Algoritmos como H-ARC, SCMBP-SCCW, LLRU e AM-LRU combinam as duas abordagens acima e dividem o *buffer* em quatro regiões: *cold-clean*, *cold-dirty*, *hot-clean* e *hot-dirty*. Assim, é possível uma melhor descrição das características de cada página.

Em 2018, o LeCaR, e posteriormente em 2021, o Cacheus, apresentaram soluções que exploram o conceito de especialistas introduzido pela ACME em 2002. Outro destaque, no de 2022, para o algoritmo de substituição baseada na abordagem Seq2Seq. O uso de estratégias de aprendizado de máquina pode se tornar uma prática comum nos algoritmos desenvolvidos nos próximos anos, principalmente se a complexidade do novo algoritmo faz com que ele trabalhe de forma constante.

## 3.4 Análise comparativa

A Tabela 1 e Tabela 2 mostram uma comparação qualitativa de algumas das políticas de substituição de *buffer* não assimétrica e assimétrica, respectivamente, discutidas neste trabalho. Os seguintes atributos foram selecionados como critérios para a análise:

- *Scan Resistance*: O recurso de política de substituição de resistir a longos acessos sequenciais que inundam todo o *buffer* e removem páginas frequentes (ver Seção 2.3.1).
- *Low Lifetime Page Protection*: A política de substituição evita que novas páginas sejam imediatamente selecionadas como vítimas.
- *Page Frequency Levels*: Quando a política de substituição apresenta diferentes níveis de diferenciação de páginas frequentes, por exemplo, os contadores de referência de página ou as listas múltiplas no algoritmo MQ (ver Seção 3.1.7).
- *Aging Mechanism*: Quando a política de substituição possui um mecanismo que evita que páginas frequentes residam no *buffer* por muito tempo quando não são acessadas por um período prolongado.
- *Tuning Parameters*: Quando a política de substituição precisa de parâmetros, por exemplo, o tamanho das listas ou mecanismos de ajuste.
- *History Mechanism*: Quando a política de substituição se beneficia de algum mecanismo que armazena um histórico de páginas acessadas anteriormente, por exemplo, páginas fantasmas (ver Seção 2.3.3).
- *Constant Complexity*: Quando a política de substituição tem uma complexidade computacional constante, independentemente do cenário da carga de trabalho.
- *Cache Starvation Issues*: Quando a política de substituição apresenta o problema do *cache starvation* (ver Seção 2.3.2).
- *Clustered Writes*: Quando a política de substituição realiza o despejo de páginas de escrita usando um método de agrupamento *(clustering)* (ver Seção 3.2.4).

Na Tabela 1 temos à análise comparativa das políticas não assimétricas, o atributo *Clustered Writes* não é aplicável, pois essas estratégias não diferenciam se o tipo de operação é escrita ou leitura durante a tomada de decisão. Outro ponto é que embora algumas dessas políticas dividam o *buffer* em regiões ou listas, nenhuma dessas estratégias apresentadas na Tabela 1 tem problemas de *cache starvation*.

Tabela 1 – Análise comparativa das políticas não assimétricas

| Política | Scan Resistance | Page Low Lifetime Protection | Page Frequency Levels | Aging Mechanism | Tuning Parameters | History Mechanism | Constant Complexity |
|---|---|---|---|---|---|---|---|
| **LRU** | Não | Sim | Não | N.A. | Não | Não | Sim |
| **LRU-MIS** | Sim | Sim | Não | N.A. | localização do ponto médio | Não | Sim |
| **FIFO** | Não | Sim | Não | N.A. | Não | Não | Sim |
| **Clock** | Não | Sim | Não | N.A. | Não | Não | Não |
| **GCLOCK** | Sim | Sim | Sim | Sim | IC e HC | Não | Não |
| **LFU** | Sim | Não | Sim | Não | Não | Não | Não |
| **LFU-DA** | Sim | Parcial | Sim | Sim | Não | Não | Não |
| **FBR** | Sim | Sim | Sim | Sim | tamanhos das listas, Amax e Cmax | Não | Sim |
| **LRU-K** | Não | Sim | Sim | Sim | K e CRP | Sim | Não |
| **2Q** | Sim | Sim | Não | N.A. | Kin e Kout | Sim | Sim |
| **LRFU** | Sim | Parcial | Sim | Parcial | $\lambda$ | Não | Não |
| **MQ** | Sim | Sim | Sim | Sim | M, Qout e LIFETIME | Sim | Sim |
| **LIRS** | Sim | Sim | Não | N.A. | LIRs e HIRs | Sim | Não |
| **ARC** | Sim | Parcial | Não | N.A. | Não | Sim | Sim |
| **CAR** | Sim | Parcial | Não | N.A. | Não | Sim | Não |
| **CART** | Sim | Parcial | Não | N.A. | Não | Sim | Não |
| **LeCaR** | Parcial | Parcial | Sim | Parcial | learning rate e discount rate | Sim | Não |
| **Cacheus** | Sim | Parcial | Sim | Parcial | Não | Sim | Não |
| **Seq2Seq** | Parcial | Parcial | Não | Parcial | threshold | Sim | Não |

Fonte: O Autor.

A Tabela 2 é uma análise comparativa de políticas assimétricas, e nenhum dos algoritmos apresentou o atributo *Page Frequency Levels* com exceção de LLRU e AM-LRU. Os demais algoritmos não usam contadores de frequência. Quando uma página recebe um *hit* ela é movida para uma região protegida (por exemplo o lado MRU) ou então o *cold bit* é atualizando, dependendo do algoritmo.

LLRU e AM-LRU, dentre todos os algoritmos analisados na Tabela 2, são os únicos aplicáveis ao recurso *Aging Mechanism*. Em ambos, cada página possui um contador que é usado para escolher a vítima em uma das quatro regiões, a fim de calcular o custo da página

Tabela 2 – Análise comparativa das políticas assimétricas

| Política | Cache Starvation | Scan Resistance | Page Low Lifetime Protection | Tuning Parameters | History Mechanism | Constant Complexity | Clustered Writes |
|---|---|---|---|---|---|---|---|
| **CFLRU** | Não | Não | Sim | W | Não | Não | Não |
| **LRU-WSR** | N.A. | Não | Sim | Não | Não | Não | Não |
| **CCF-LRU** | Sim | Parcial | Parcial | Não | Não | Não | Não |
| **CCCF-LRU** | Não | Parcial | Sim | minCCL | Não | Não | Não |
| **CFDC** | Sim | Não | Sim | priority window e tamanho do cluster | Não | Não | Sim |
| **CASA** | Não | Parcial | Parcial | $C_R$ e $C_W$ | Não | Sim | Não |
| **AD-LRU** | Não | Sim | Sim | min_lc | Não | Não | Não |
| **SCMBP-SCCW** | Não | Sim | Parcial | valores de aumento das regiões e tamanho do cluster | Sim | Não | Sim |
| **GASA** | Sim | Parcial | Parcial | GSMIN e GSMAX | Sim | Não | Não |
| **LLRU** | Sim | Parcial | Parcial | $E_{CC}$ e $E_{CD}$ | Não | Sim | Não |
| **AM-LRU** | Não | Sim | Sim | $E_{CC}$, $E_{CD}$ e min_len | Não | Não | Não |

Fonte: O Autor.

*(PC)*. Portanto, se uma página receber muitos acessos, é notável que ela permanecerá no *buffer* por muito tempo, pois ambos os algoritmos não possuem mecanismo de envelhecimento.

### *3.4.1 Scan resistance*

Analisamos o mecanismo de *scan resistance* nas políticas de substituição de *buffer*. Durante o acesso sequencial, os algoritmos sofrem uma série de falhas *(miss)* de requisições que podem estimular a adaptação do algoritmo para minimizar o impacto negativo desse processamento. Embora muitos dos algoritmos consigam manter páginas frequentes após sofrerem acesso sequencial, não é garantido que a classificação dessas páginas frequentes seja adequada para processamento futuro. Portanto, quanto mais rápido o algoritmo identificar o acesso sequencial e quanto menos páginas importantes forem despejadas, menos significativo será o impacto negativo.

Algoritmos como LRU, FIFO, *Clock* e LRU-K não possuem mecanismos adequados

para evitar que o acesso sequencial à página inunde todo o *buffer*, removendo assim as páginas frequentes e reduzindo diretamente o desempenho. Enquanto nos algoritmos FIFO e LRU, a varredura sequencial desloca todas as páginas para o final da lista. O algoritmo *Clock* dá uma segunda chance para cada página, esse atraso ainda não é o suficiente para evitar que todas as páginas sejam despejadas. Um processo semelhante também ocorre no LRU-K.

Um contraponto são os algoritmos LRU-MIS, 2Q, MQ, ARC, CAR e CART que resistem aos acessos sequenciais. No entanto, eles têm estruturas e mecanismos diferentes, mas implementam a mesma ideia: dividir o *buffer* em regiões para proteger páginas frequentes. Assim, essas estratégias limitam uma região a manter apenas novas páginas e outra região ou regiões, no caso do MQ, a manter páginas frequentes. Por exemplo, o algoritmo MQ divide o *buffer* em vários níveis de frequência de página. Novas páginas são inseridas no último nível. Assim, em um cenário de acessos sequencial deve afetar mais intensamente as páginas do último nível. Portanto, o algoritmo atrasa a remoção das páginas frequentes de níveis mais altos.

Outro mecanismo de *scan resistance* é o algoritmo LIRS. As novas páginas são classificadas como HIR. Apesar de estarem inseridas nas duas regiões, as novas páginas do HIR afetam diretamente apenas a lista Q, da qual serão escolhidas as próximas vítimas. O LIRS mantém as páginas LIR do *buffer*, o que impede que sejam despejadas durante essas operações sequenciais.

Os algoritmos GCLOCK, LFU, LFU-DA e FBR mantêm um contador de referência para cada página. Essas estratégias primeiro despejam as páginas com os contadores de referência mais baixos durante o acesso sequencial, protegendo assim as páginas mais frequentes de acordo com a classificação do algoritmo. O algoritmo LRFU funciona de forma semelhante, mas usa o valor CRF para cada página. Quando o parâmetro $\lambda$ se aproxima de 0, o valor do CRF de uma página é incrementado aproximadamente em 1 após um acerto, assumindo as características do LFU. Por outro lado, quando o valor de $\lambda$ se aproxima de 1, o algoritmo tende a despejar a página LRU.

LeCaR é baseado na ideia dos múltiplos especialistas; Ele executa os algoritmos LRU e LFU simultaneamente. O mecanismo de *scan resistance* vai variar e pode assumir o comportamento de um LRU ou LFU dependendo da carga de trabalho e dos processos de aprendizado. No entanto, o algoritmo CACHEUS melhora os especialistas em LeCaR, introduzindo mecanismos sofisticados para lidar com acessos sequenciais.

Para a política de substituição baseada na abordagem Seq2Seq, dependendo da

previsão do modelo, o algoritmo pode se comportar com *scan resistance*, pois a previsão pode indicar que páginas recentes e classificá-las como candidatas a vítimas.

Para as políticas assimétricas na Tabela 2, algoritmos como CFLRU, LRU-WSR e CFDC não protegem de um acesso sequencial e pode remover as páginas frequentes do *buffer*. No entanto, consideramos um *scan resistance* parcial para CCF-LRU, CCCF-LRU, CASA, GASA e LLRU. Estas abordagens podem proteger as páginas *dirty* durante o acesso sequencial enquanto penalizam as páginas *clean*. Finalmente, algoritmos como AD-LRU, SCMBP-SCCW e AM-LRU lidam com o problema dos acessos sequenciais protegendo páginas *clean* e *dirty* movendo-as para regiões específicas.

### *3.4.2 Page low lifetime protection*

Determinar o tempo de vida que uma página recente deve permanecer no *buffer* é um desafio no desenvolvimento de políticas sofisticadas de substituição. De maneira otimista, os algoritmos atrasam a substituição das páginas recentes para esperar que essas páginas recebam acessos futuros. Por outro lado, o tamanho do *buffer* é limitado, fazendo com que as páginas recentes afetem diretamente o número de páginas frequentes atuais no *buffer*.

Algoritmos como LRU, LRU-MIS, FIFO, FBR, 2Q e LIRS controlam o tempo de vida quando deslocam as páginas nas suas listas até que cheguem ao final da lista. O algoritmo *Clock* controla o tempo de vida quando dá uma segunda chance às novas páginas usando os bits de referência e movendo o ponteiro da página. Para esses tipos de algoritmos, o tempo de vida mínimo aproximado de uma nova página depende do tamanho do *buffer* ou dos parâmetros, por exemplo, o ponto médio do algoritmo LRU-MIS.

Em um cenário onde um conjunto expressivo de páginas é classificado como frequente e residem no *buffer* do algoritmo LFU, as novas páginas podem ter um tempo de vida curto ou, em alguns casos, ser imediatamente despejadas na requisição subsequente. No caso do algoritmo LFU-DA, quando no mesmo cenário, o mecanismo de envelhecimento tende a tratar o problema parcialmente. Após um número significativo de requisições, as páginas anteriormente classificadas como frequentes são despejadas, tendo mais espaço no *buffer* para as novas páginas. Um processo semelhante ocorre com os algoritmos LRFU, LeCaR e CACHEUS, essas estratégias podem se comportar de maneira semelhante ao LFU porque depende da carga de trabalho, do parâmetro $\lambda$ no LRFU ou do processo de aprendizado no LeCaR e CACHEUS. A política de substituição baseada na abordagem Seq2Seq se comporta parcialmente porque o mecanismo de

previsão pode influenciar as decisões e remover uma página recente.

Os algoritmos ARC, CAR e CART têm tamanhos de regiões de frequência e recência dinâmicas. Assim, em cenários específicos, essas estratégias podem limitar significativamente o tamanho da região de recência e, consequentemente, diminuir o tempo de vida das novas páginas. Outras estratégias, como GCLOCK, LRU-K e MQ, usam parâmetros que determinam o tempo de vida mínimo aproximado de novas páginas.

Para políticas assimétricas na Tabela 2, uma das principais características desses algoritmos é que as páginas *clean* tendem a ser despejadas primeiro, aumentando a tempo de vida das páginas *dirty*. Por exemplo, o algoritmo LRU-WSR dá às páginas *dirty* uma segunda chance usando uma estratégia semelhante ao algoritmo Clock.

Algoritmos como CFLRU, CCCF-LRU, CFDC, AD-LRU, AM-LRU apresentam um tempo de vida paras as páginas *clean* e *dirty* semelhante ao algoritmo LRU. As páginas residem no *buffer* em uma região específica e quando novas páginas são solicitadas, as páginas são deslocadas para outras regiões ou substituídas. Por outro lado, os algoritmos CCF-LRU e GASA apresentam proteção parcial. As páginas *clean* são substituídas imediatamente ou com um tempo de vida baixo em cenários específicos. No entanto, as páginas *dirty* devem ter um tempo de vida significativo, pois essas estratégias implementam uma segunda chance para páginas *dirty* semelhante ao algoritmo LRU-WSR.

Os algoritmos SCMBP-SCCW, CASA e LLRU controlam dinamicamente o tamanho de suas regiões. Assim, semelhante ao ARC, as novas páginas podem ter um tempo de vida baixo, independentemente das páginas *clean* ou *dirty* em cenários específicos.

### *3.4.3 Page frequency levels*

Cada página pode receber uma série de acessos tornando-se frequentes durante as transações. Cada algoritmo lida de uma forma para classificar melhor as páginas frequentes. O algoritmo LRU apenas move as páginas para o lado mais recente da lista e não há mecanismo para saber se uma página recebeu mais visitas do que outra. Em contraste, o algoritmo LFU mantém um contador que classifica quais páginas receberam mais acessos. Quando um algoritmo classifica diferentes níveis de frequência para cada página, é possível aproveitar um novo critério para escolher uma página vítima.

Algoritmos como GCLOCK, LFU, LFU-DA, FBR, LRFU, LeCaR e CACHEUS diferenciam páginas usando contadores de referência ou, no caso de LRFU, o cálculo da CRF

durante o processo de seleção da vítima. Os algoritmos MQ e LRU-K se destacam na diferenciação de páginas com abordagens sofisticadas sem contadores. O algoritmo MQ divide o *buffer* em listas de páginas de tamanho dinâmico que representam cada nível de frequência. O algoritmo LRU-K mantém um histórico com o tempo das referências de cada página. Portanto, quanto maior o tamanho do parâmetro K, a tendência é que o algoritmo diferencie melhor as páginas frequentes.

Algoritmos como LRU, FIFO e LRU-MIS não possuem nenhum mecanismo para diferenciar páginas frequentes. Para os algoritmos Clock, LIRS, CAR e CART usam bits de referência que classificam as páginas em dois níveis. No entanto, não consideramos uma abordagem eficiente na diferenciação de páginas frequentes. Isso também ocorre com os algoritmos ARC e 2Q que apenas diferenciam se as páginas são recentes ou frequentes.

### *3.4.4 Aging mechanism*

Não consideramos mecanismos de envelhecimento aplicáveis a algoritmos na Tabela 2 que não possuem níveis de frequência de página. Uma vez que essas estratégias removem sem esforço as páginas frequentes quando não há mais acessos. Por exemplo, páginas frequentes que não recebem mais ocorrências no algoritmo LRU são deslocadas para o final da lista à medida que novas solicitações chegam. O algoritmo LFU não possui um mecanismo de envelhecimento. Mesmo que uma página frequente pare de obter acessos, o LFU pode levar muito tempo para remover essa página frequente, dependendo da carga de trabalho. Este problema foi minimizado quando o algoritmo LFU-DA implementou seu mecanismo de envelhecimento com o fator de inflação (veja a Seção 2.2).

Os algoritmos GCLOCK e MQ implementam mecanismos de envelhecimento de uma maneira particular porque aproveitam suas estruturas de dados. O algoritmo GCLOCK usa a lista circular e diminui os contadores de referência de página quando o ponteiro do relógio gira. O mecanismo de envelhecimento do algoritmo MQ usa a estrutura de várias filas para rebaixar páginas frequentes para filas inferiores quando o tempo de vida expirar. Por fim, essa página frequente quando não recebe mais acessos deve ser movida para a última fila inferior e ser despejada.

O mecanismo de envelhecimento FBR é baseado na estratégia de *aging by division* (EFFELSBERG; HäRDER, 1984). O FBR reduz os contadores de referência aplicando a divisão quando a média dos contadores de referência excede o parâmetro $A_{max}$.

O algoritmo LRU-K usa o histórico de acesso (faixas) para cada página. Essa abordagem muda os valores históricos associados a cada página, extinguindo as referências mais antigas. Quando uma página frequente deixa de receber *hits* e novas páginas são inseridas, sua referência mais antiga é envelhecida e eventualmente escolhida como vítima.

Quando o parâmetro $\lambda$ do algoritmo LRFU se aproxima de 1, o algoritmo se degenera para se comportar de forma semelhante ao LRU. Portanto, nenhum mecanismo de envelhecimento é aplicável. No entanto, quando $\lambda$ se aproxima de 0, o algoritmo se comporta de forma semelhante ao LFU, ele pode ter dificuldades para substituir páginas antigas frequentes. Quando os valores de CRF são recalculados, esse impacto é minimizado.

Os algoritmos LeCaR e CACHEUS usam a estratégia de especialistas. Mesmo que um dos especialistas tenha problemas para substituir páginas antigas frequentes, o outro influenciará diretamente nas decisões devido aos processos de aprendizado.

Para a política de substituição baseada na abordagem Seq2Seq, são feitas novas predições que podem eventualmente influenciar a remoção de páginas frequentemente não utilizadas. No entanto, pode ocorrer o contrário, e o algoritmo pode manter essas páginas frequentes e antigas no *buffer* por muito tempo, pois podem impactar significativamente o mecanismo de previsão.

### *3.4.5 Tuning parameters*

Os parâmetros de ajuste podem afetar diretamente o tamanho das estruturas de dados do algoritmo, por exemplo, LRU-MIS, FBR, 2Q, MQ, LRFU e LIRS. Os autores podem sugerir valores para os parâmetros por meio de testes rigorosos em diferentes cargas de trabalho. Porém, na prática, esses valores são aproximações do valor ideal que busca atender diferentes tipos de cargas de trabalho simultaneamente.

Um dos benefícios dos algoritmos que não usam parâmetros de ajuste é que eles são facilmente escaláveis com diferentes tamanhos de *buffer*. No caso do algoritmo ARC, ele também pode se adaptar a diferentes cargas de trabalho sem a necessidade de parâmetros. No entanto, pode haver um alto custo até que o algoritmo possa identificar um padrão. Por exemplo, o ARC pode precisar remover páginas importantes até identificar o padrão durante um acesso sequencial.

Algoritmos como GCLOCK, LRU-K e MQ usam parâmetros de ajuste para controlar o tempo de vida das novas páginas. O algoritmo LeCaR utiliza parâmetros para gerenciar o

processo de aprendizagem. O CACHEUS também aprimora o algoritmo LeCaR neste aspecto usando uma abordagem de adaptação de parâmetros.

A política de substituição baseada no Seq2Seq conta o número de falhas de predição e quando atingem um limite, faz novas predições. O valor limite influencia diretamente o desempenho do *buffer*. Uma vez que, se o tamanho for muito pequeno, o algoritmo precisa fazer mais predições.

Na Tabela 2 os algoritmos CCCF-LRU e AM-LRU usam um parâmetro de ajuste que define um tamanho mínimo para suas regiões de páginas novas *clean*. Um parâmetro semelhante é usado no AD-LRU, mas para páginas *clean* e *dirty*. Esse parâmetro pode ser bastante sensível para essas estratégias. Quanto maior, essas páginas terão um tempo de vida significativo, mas podem ocupar espaço de *buffer* que poderia ser usado para armazenar páginas frequentes ou *dirty*.

Compreender o impacto no custo de escrita ou leitura é fundamental na concepção de políticas assimétricas. Estratégias como CASA, LLRU e AM-LRU aproveitam essas informações usando parâmetros de ajuste que definem esses custos. Dependendo desses custos, o comportamento desses algoritmos pode ser mais flexível, adaptando-se melhor à assimetria da mídia de armazenamento.

Os algoritmos CFLRU e CFDC usam parâmetros de ajuste para definir os tamanhos das suas estruturas *working region* e *priority region*, respectivamente. No entanto, a CFLRU também pode variar este parâmetro dinamicamente. O algoritmo GASA usa parâmetros de ajuste para definir os limites de tamanho de seu mecanismo de histórico. Um histórico extenso pode não refletir o comportamento real das páginas solicitadas anteriormente.

O tamanho do *cluster* do mecanismo de agrupamento de páginas *dirty* dos algoritmos CFCD e SCMBP-SCCW é definido por meio do parâmetro de ajuste. O algoritmo SCMBP-SCCW também usa vários parâmetros de ajuste que controlam o tamanho das regiões. Esses parâmetros são extremamente sensíveis porque tratam do equilíbrio entre escrita e leitura (assimetria) e ao mesmo tempo a recência e a frequência.

### *3.4.6 History mechanism*

Algoritmos que utilizam um histórico de requisições passadas podem trazer diversos benefícios como a adaptação e detecção de padrões, mitigar o impacto negativo da substituição imediata de novas páginas ou uma melhor classificação de páginas frequentemente descartadas

quando acessadas novamente. A implementação do histórico deve ter uma atenção redobrada, pois impacta no consumo de memória ou no desempenho do *buffer* ao buscar as páginas.

Algoritmos como 2Q, MQ e LIRS melhoram a identificação de páginas frequentes substituídas anteriormente que residem no histórico. Os algoritmos ARC, CAR e CART fazem o mesmo e alteram dinamicamente o tamanho de suas estruturas de dados. LeCaR e CACHEUS usam os acessos às páginas do histórico para atualizar os pesos dos especialistas essenciais nas decisões das vítimas. Na mesma linha, o LRU-K classifica melhor as páginas frequentes e aplica o mecanismo de envelhecimento com histórico. Por fim, a política de substituição baseada no Seq2Seq mantém um histórico de solicitações utilizadas como entrada para o mecanismo de previsão.

Para as políticas assimétricas da Tabela 2, apenas os algoritmos SCMBP-SCCW e GASA fazem uso dos mecanismos de histórico. Ambos usam uma estrutura de lista, SCMBP-SCCW tem como base o algoritmo ARC, porém usa quatro listas históricas associadas para cada região. O algoritmo GASA simplifica usando apenas uma lista para histórico.

### *3.4.7 Constant Complexity*

A complexidade do algoritmo é um dos aspectos essenciais da implementação de políticas de substituição de *buffer*. O gerenciador de *buffer* recebe uma vasta sobrecarga de processamento (HARIZOPOULOS *et al.*, 2008). Portanto, algoritmos com complexidade constante tendem a ser uma alternativa mais propensa.

O algoritmo LRU tem uma implementação simples e uma complexidade constante. Variações do algoritmo LRU como LRU-MIS são comuns em implementações de banco de dados (ORACLE, 2021a). Outros sistemas optam por implementar variantes do algoritmo *Clock*, embora não tenha uma complexidade constante devido ao processo de seleção de vítimas, a mecânica de ponteiro e a estrutura de dados circular facilitam o processamento de transações simultâneas (XIA; BU, 2015).

Algoritmos como LFU, LFU-DA, LRU-K e LRFU não apresentam complexidade constante porque seus conceitos refletem em implementações com estruturas mais complexas, como *heap*. O mesmo ocorre para algoritmos com a estratégia de especialistas LeCaR e CACHEUS que utilizam o algoritmo LFU ou variações. E também para a política de substituição baseada no Seq2Seq, que utiliza uma estrutura de predição mais complexa baseada em redes neurais.

Na Tabela 2, destacamos os algoritmos CASA e LLRU com complexidade constante. Em cenários específicos, o algoritmo AM-LRU não apresenta complexidade constante. Embora se baseie no LLRU, a abordagem de seleção de vítimas é diferente e consiste em dar uma segunda chance às páginas candidatas de vítimas das regiões quentes, movendo-as para as regiões frias. Portanto, uma vítima sempre será escolhida de uma região fria. Este procedimento pode ser repetido. Os demais algoritmos não apresentam uma complexidade constante devido aos seus mecanismos de segunda chance, busca por páginas *clean* que preferem ser removidas primeiro ou mecanismos de agrupamento.

### *3.4.8 Cache Starvation*

O problema de *cache starvation* faz com que algumas regiões definidas pelo algoritmo permaneçam pequenas devido a alguma priorização de substituição de páginas nessa região. Isso pode afetar diretamente o *hit ratio* quando a carga de trabalho muda constantemente e as que teriam benefício em manter essas páginas estão comprometidas.

O algoritmo LRU-WSR não se aplica ao problema do *cache starvation* porque possui apenas uma estrutura de lista. No entanto, algoritmos como CCF-LRU, CFDC, GASA e LLRU não possuem mecanismos para evitar o problema. Essas estratégias podem afetar negativamente o desempenho quando as regiões que mantêm as páginas *clean* podem ficar muito pequenas.

Os algoritmos CCCF-LRU e AM-LRU corrigem o problema do *cache starvation* dos algoritmos CCF-LRU e LLRU, respectivamente. A abordagem usada é a mesma do AD-LRU, usando um parâmetro de ajuste que define um tamanho mínimo para a região de páginas *clean*.

### *3.4.9 Clustered Writes*

Na Tabela 2 apenas os algoritmos CFDC e SCMBP-SCCW usam mecanismos de agrupamento. Embora usem uma técnica semelhante, há uma diferença em qual região o mecanismo de agrupamento é aplicado.

No algoritmo CFDC, as páginas *dirty* são inseridas primeiro na *working region* e permanecem por um tempo de vida significativo para serem posteriormente movidas para a região dos *clusters* e por fim são despejadas. Uma desvantagem é que no caso de um acesso sequencial de páginas *dirty*, o mecanismo de agrupamento deve demorar para agir porque as páginas sequenciais devem inundar a *working region* primeiro e podem remover páginas frequentes.

O algoritmo SCMBP-SCCW agrupa diretamente as páginas *dirty* na sua região de recência. O ponto positivo é que o mecanismo de agrupamento atua imediatamente com menos risco de remover páginas frequentes de outras regiões durante um acesso sequencial de páginas *dirty*. Uma desvantagem é que as páginas *dirty* recentes podem ter uma vida útil baixa em cenários específicos, pois são inseridas em um *cluster* de baixa prioridade, por exemplo.

## 3.5 Conclusão

Este capítulo introduziu diversas políticas de substituição classificadas como estratégias para mídias assimétricas e não assimétricas. Ao logo das Seções, focamos em apresentar o fluxo de execução destes algoritmos e suas principais características. Em seguida, uma linha do tempo histórica foi elaborada e junto com uma análise comparativa dessas principais estratégias.

# 4 PROPOSTA

Neste capítulo é proposto uma estratégia de substituição de páginas em *buffer* para banco de dados denominada de EBRES *(Efficient Buffer Replacement with Exponential Smoothing)*. O gerenciador de *buffer* é um componente com um impacto computacional significativo em SGBD. Técnicas de aprendizagem de máquina que executam múltiplos algoritmos simultaneamente (VIETRI *et al.*, 2018) (RODRIGUEZ *et al.*, 2021) ou redes neurais (CHOI; PARK, 2022) apresentam complexidade computacional significativa. No contexto de gerenciamento de *buffer*, é fundamental que as políticas de substituição tenham uma complexidade computacional baixa e também considerem a assimetria da mídia de armazenamento secundária.

EBRES é uma política inteligente de substituição de páginas em *buffer*. A ideia é fazer uso do modelo de predição de suavização exponencial com uma baixa sobrecarga no sistema para tentar identificar páginas importantes no *buffer*, melhorando o *hit ratio*. A suavização exponencial foi usada em (LEVANDOSKI *et al.*, 2013), no contexto de bancos de dados em memória principal para classificar dados (frios e quentes) predizendo o seu número de acessos (frequência). No caso específico do EBRES, a ideia é utilizar o modelo para predizer quando será o próximo acesso de uma página *P* e, com isso, basear a decisão de evicção. EBRES também lida com mídia assimétrica, buscando reduzir o número de escritas.

A suavização exponencial será abordada com mais detalhes na próxima Seção 4.1. Contudo, apresenta vantagens que podem ser que relevantes para o contexto de *buffer* como uma complexidade constante de execução e suas predições possuem como resultado com uma influência maior das últimas entradas de dados usadas no treinamento. Além disso, outros trabalhos baseados no trabalho de (LEVANDOSKI *et al.*, 2013) também usam a suavização exponencial:

- O trabalho de (STOICA; AILAMAKI, 2013) propõe uma técnica de baixa sobrecarga para migrar dados frios para armazenamento secundário de banco de dados em memória implementada no VoltDB (VOLTACTIVEDATA, 2010) um SGBD comercial *open-source*.
- O projeto Siberia (ELDAWY *et al.*, 2014) apresenta um *framework* capaz de gerenciar os dados frios gerados pelo Microsoft Hekaton (DIACONU *et al.*, 2013) uma *engine* de banco de dados em memória otimizada para cargas de trabalho OLTP.
- O trabalho de (ZHANG *et al.*, 2015) usa a suavização exponencial para identificar triplas[1]

[1] Uma base de dados em RDF consiste em um conjunto de triplas. Cada tripla pode ser interpretada como uma afirmação sobre um recurso. A tripla possui três elementos e o formato utilizado é: sujeito, predicado e objeto.

quentes com o objetivo de melhorar a performance das consultas em dados RDF. Os experimentos foram conduzidos em um *endpoint* SPARQL.

## 4.1 O modelo de suavização exponencial no gerenciamento de *buffer*

As requisições geradas aos gerenciadores de *buffer* podem ser representadas por uma série temporal. Em diversos casos, existe uma relação entre páginas requisitadas e o tempo de requisição. Algoritmos como o LRU-K e LIRS (ver Seções 3.1.2 e 3.1.8) exploram essas relações na elaboração das suas estratégias. De maneira semelhante, os modelos de predição de séries temporais geralmente se baseiam em dados históricos e inter-relacionamentos para suas predições de valores futuros (SORJAMAA *et al.*, 2007).

A estratégia proposta por (CHOI; PARK, 2022) (ver Seção 3.1.12) utiliza um modelo preditivo com foco na abordagem Seq2Seq, embora possa demonstrar resultados eficientes ainda apresenta uma complexidade computacional significativa por se tratar de uma rede neural. Por outro lado, outros algoritmos usam modelos mais simples, por exemplo na estratégia de (LEVANDOSKI *et al.*, 2013) lida com o modelo de suavização exponencial *(exponential smoothing)*. A suavização exponencial é modelo de predição de séries temporais que utiliza uma equação de médias móveis simples, porém ponderadas exponencialmente. Além disso, esse modelo é capaz de ser treinado apenas com as observações da própria série de dados (VERÍSSIMO *et al.*, 2013).

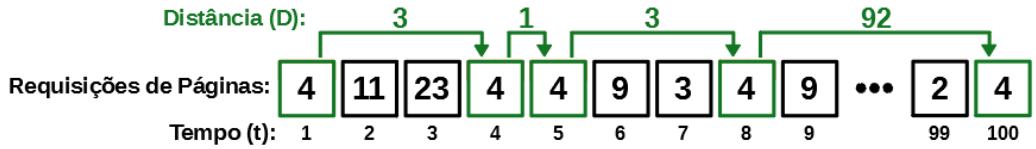
$$S_t = \alpha x_t + (1-\alpha)S_{t-1} = S_{t-1} + \alpha(x_t - S_{t-1}) \tag{4.1}$$

Há diversos métodos de suavização exponencial, como a suavização exponencial simples, dupla (método de Holt), e tripla (método de Holt-Winters). Neste trabalho, vamos focar com o método simples usando a Fórmula 4.1 com as seguintes definições (JR, 2006).

- $S_t$: Representa o valor esperado da predição e é calculado após a primeira observação.
- $x_t$: Representa o valor observado da série temporal no período $t$.
- $S_{t-1}$: Representa o valor da predição calculada anteriormente.
- $\alpha$: Representa o fator de suavização, que vária entre 0 e 1. É um parâmetro de ajuste usado para calibrar a suavização.

Na posição do sujeito, está o recurso sobre qual a afirmação está sendo feita. O predicado é outro recurso que denota uma propriedade do sujeito e o relaciona através dessa propriedade com o objeto. O objeto pode, por sua vez, ser outro recurso ou possuir um valor literal, como um número ou uma sequência de caracteres.

Figura 29 – Exemplo com distâncias em um conjunto de requisições de páginas

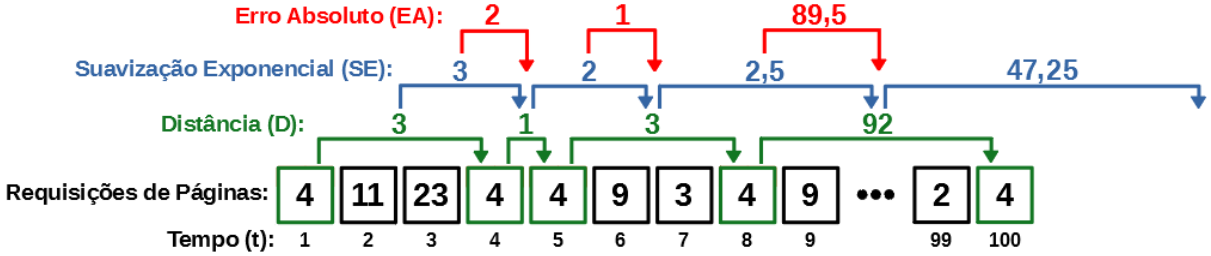



Fonte: O Autor.

A Figura 29 exemplifica um conjunto de requisições de páginas feita a um gerenciador de *buffer*. Selecionando apenas a página de identificador 4, que aparece nos tempos 1, 4, 5, 8 e 100. Podemos calcular as distâncias (D) entre as requisições subsequentes da página 4. Intuitivamente, poderíamos realizar uma média simples dos valores de distância (3, 1, 3 e 92) com a intenção de predizer qual será a distância da próxima requisição futura da página 4. Porém, no contexto de gerenciamento de *buffer* vários fatores podem interferir na ordem das requisições, como o processamento das transações de forma concorrente. Isto é, partimos da premissa que requisições passadas podem não ter nenhuma influência para as requisições futuras. E por conta disso, usar apenas a média pode não trazer resultados precisos.

Em vez de usar uma média simples, podemos utilizar o modelo de suavização exponencial que é capaz de lidar com o conjunto de maneira ponderada para cada distância. É possível então considerar que as distâncias mais recentes recebam um peso maior, que declina exponencialmente até as distâncias mais antigas. Logo, as requisições muito antigas da página perdem uma parte da sua influência na predição, enquanto as requisições novas se sobressaem.

Figura 30 – Exemplo de suavização exponencial com um conjunto de requisições de páginas




Fonte: O Autor.

Usando o mesmo exemplo da Figura 29, vamos incluir na Figura 30 um exemplo da predição de distâncias de páginas usando suavização exponencial. A Formula 4.1 foi adaptada para ser usada nesse contexto: $SE_t = SE_{t-1} + \alpha(D_t - SE_{t-1})$, onde $D_t$ representa a distância e o fator $\alpha$ tem o valor de 0,5. O modelo é aplicado para a página 4 nos tempos 1, 4, 5, 8, e 100.

- No tempo 1: Não é possível calcular o $SE_1$, pois não temos outra requisição para calcular

a distância da página 4.

- No tempo 4: Como não foi calculado nenhuma predição anterior ($SE_{t-1}$) vamos assumir para esse contexto aplicado que o valor da predição será igual a distância ($SE_4 = 3$).
- No tempo 5: Temos a distância $D_5 = 1$ e a predição anterior ($SE_4$) assumimos que é 3. Logo, podemos aplicar o modelo $SE_5 = 3 + \alpha(1-3) = 2$.
- No tempo 8: Temos a distância $D_8 = 3$ e a predição anterior $SE_5 = 2$ . Logo, podemos aplicar o modelo $SE_8 = 2 + \alpha(3-2) = 2,5$
- No tempo 100, temos a distância $D_{100} = 92$ e a predição anterior $SE_8 = 2,5$. Logo $SE_{100} = 2,5 + \alpha(92-2,5) = 47,25$.

Quando aplicamos o modelo de suavização exponencial em uma série temporal gerada individualmente para cada página podemos encontrar uma estimativa de quando essa página será acessada no futuro próximo. A execução do método de suavização exponencial simples possui uma baixa sobrecarga devido a complexidade constante, sendo capaz de ser recalculada a cada acesso à página no *buffer*. Podemos avaliar o erro das predições usando o método do erro absoluto *(EA)*, subtraindo o último valor predito com a distância atual $EA = |SE_{t-1} - D_t|$. Como o valor de *SE* no tempo 100 é grande, isto significa que a página não foi muito acessada ultimamente. Para o contexto de gerenciamento de *buffer* as páginas menos recentes com predições e erros maiores podem ser vistas como candidatas a vítimas primeiramente. A frequência de acesso também pode ser considerada, páginas com uma baixa frequência possuem uma probabilidade maior de que suas predições estejam incorretas, pois não tiveram dados o suficiente para montar uma série temporal maior.

## 4.2 Estruturas de dados do EBRES

Diversas configurações das estruturas de algoritmos de substituição foram discutidas nas Seções 3.1 e 3.2. Maioria desses algoritmos usam estruturas simples como listas para dividir o *buffer* em regiões. Várias configurações para mídias assimétricas dividem o *buffer* em listas contendo páginas *clean* ou *dirty*, assim podem facilitar a remoção das páginas *clean* primeiramente. Por outro lado, algoritmos não assimétricos como o ARC, tem uma maior preocupação em separar as páginas recentes (conhecidas como *cold* em algumas estratégias) das páginas frequentes (também chamadas de *hot* em algumas estratégias).

Existem também aqueles algoritmos que buscam um maior equilíbrio entre as duas configurações como por exemplo o H-ARC, SCMBP-SCCW, LLRU e AM-LRU. EBRES também

adota a mesma configuração de listas inspiradas nesses algoritmos, porém usando apenas duas listas *ghost* auxiliares:

- *cold-clean*: Mantém as páginas recentes de leitura.
- *cold-diry*: Mantém as páginas recentes de escrita.
- *hot-clean*: Mantém as páginas frequentes de leitura.
- *hot-diry*: Mantém as páginas frequentes de escrita.
- *In (ghost)*: Mantém um histórico de entrada das requisições por páginas acessadas no *buffer*, semelhante ao *History Buffer* do algoritmo de substituição baseada na abordagem Seq2Seq (ver Seção 3.1.12). É usada para computar estatísticas importantes dos padrões de acessos das últimas requisições feitas.
- *Out (ghost)*: Mantém um histórico de saída de todas as páginas vítimas das quatro listas principais na ordem de saída. É usada para recuperar metadados das páginas que foram re-acessadas, porém já haviam sido despejadas antes.

As quatro listas principais possuem tamanhos dinâmicos ajustando durante a execução do algoritmo. Já as duas listas *ghost* usam os parâmetros de ajuste *IN_SIZE* e *OUT_SIZE* definir seus tamanhos fixo. Alguns algoritmos como o CCCF-LRU, AD-LRU e AM-LRU adotam uma abordagem simples com um parâmetro de ajuste que define um tamanho mínimo para uma lista, dessa forma os algoritmos são capazes de evitar o problema do *cache starvation*. Usamos uma abordagem semelhante para o EBRES, porém usando quatro parâmetros de ajuste para cada lista principal chamados: *MIN_COLD_CLEAN*, *MIN_COLD_DIRTY*, *MIN_HOT_CLEAN*, e *MIN_HOT_DIRTY*.

Pode existir um cenário em que o *buffer* executa uma carga de trabalho com todas as requisições apenas de leitura (ou vice-versa) por um longo período de tempo. Os parâmetros *MIN_COLD_DIRTY* e *MIN_HOT_DIRTY* fariam que as duas listas de escrita nunca ficassem vazias. Nesse tipo de cenário, em que a intensão de leitura muito excessiva por um longo período, talvez não seja desejável manter essas páginas de escritas. Podemos usar o seu espaço da memória para manter mais páginas de leitura, com o objetivo de favorecer a carga de trabalho.

Por conta deste aspecto, EBRES introduz o conceito de frequência de leitura e escrita (*read_frequency* e *write_frequency*), o qual é um mecanismo fundamental para controlar o tamanho das listas de maneira dinâmica e permite que, dependendo da carga de trabalho, uma lista possa ficar vazia. EBRES usa o mecanismo de coleta de estatísticas usando o histórico *IN* para calcular as duas frequências (ver Seção 4.5). Cada uma das quatro listas terá uma variável

que determina o seu tamanho desejável sendo calculada da seguinte forma:

- $desirable\_cold\_clean\_length = MIN\_COLD\_CLEAN * read\_frequency$
- $desirable\_cold\_dirty\_length = MIN\_COLD\_DIRTY * write\_frequency$
- $desirable\_hot\_clean\_length = MIN\_HOT\_CLEAN * read\_frequency$
- $desirable\_hot\_dirty\_length = MIN\_HOT\_DIRTY * write\_frequency$

Logo, quanto mais o $read\_frequency$ ou $write\_frequency$ se aproxima de zero, menor será o tamanho desejável das duas listas correspondentes. Chegando a zero, em um caso de um cenário muito excessivo, isso não significa que teremos um problema de *cache starvation* porque a partir do momento que a carga de trabalho muda, as frequências também devem mudar, fazendo com que as listas vazias cresçam novamente.

## 4.3 Análise dos metadados

É comum em vários algoritmos como o LFU, MQ, LLRU e AM-LRU armazenar metadados nas suas páginas, por exemplo, um contador de acessos usado no processo de escolha da vítima. Vamos considerar um cenário hipotético em que temos um banco de dados que possui o tamanho do seu bloco de dados de 4096 bytes, e que consegue armazenar 100 milhões de páginas na memória. Isto resultaria um total de 409,6 gigabytes apenas dos dados do bloco. Caso quiséssemos adicionar um contador de referências com um suposto tamanho de 4 bytes, teríamos um consumo de 400 megabytes apenas desse metadado.

O consumo desse metadado não é tão significativo dado a proporção de uma base hipotética tão grande. Logo, espera-se melhorar a performance do algoritmo por meio de metadados aumentando um pouco o consumo de memória. No caso do EBRES, é preciso usar cinco metadados adicionais para cada página que reside nas quatro listas principais ou no histórico *OUT*:

- *references*: contador de acessos, semelhante ao LFU.
- *last*: armazena o tempo do último acesso dessa página.
- *list*: armazena um enumerador que indica em qual das quatro listas principais essa página reside ou então informa se reside no histórico *OUT*.
- *SE*: armazena o valor da predição do método de suavização exponencial.
- *error*: armazena o valor do cálculo do erro absoluto a partir do resultado da predição.

Os tamanhos em bytes de cada um desse metadados depende da linguagem de programação usada. Porém, como uma estimativa, vamos supor que os metadados *references* e

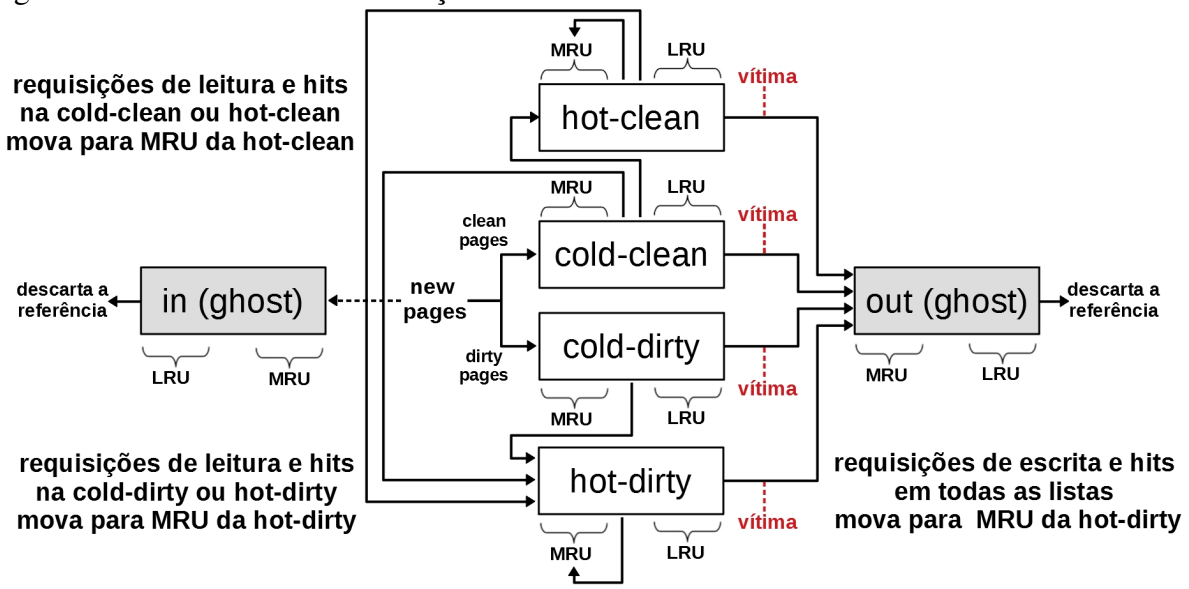
*last* são números inteiros com 4 e 8 bytes, respectivamente. Para os metadados *SE* e *error* são pontos flutuantes ambos com 4 bytes. E para o enumerador *list* adotamos um 1 byte. Teríamos um consumo de cerca de 21 bytes para cada página. No nosso exemplo hipotético, EBRES teria um consumo estimado de 2.1 gigabytes para esses metadados.

Para os metadados do histórico *OUT*, temos os mesmos cinco metadados das quatro listas principais, logo teríamos 21 * *OUT_SIZE* bytes para esses metadados. Já para o histórico *IN* temos apenas duas *flags*: *is_miss* e *is_dirty*. Podemos estimar que as duas *flags* tem um tamanho de 1 byte. Logo, teríamos um consumo estimado de 2 * *IN_SIZE* bytes. É claro além dos metadados exclusivos da estratégia EBRES, em sistemas concretos, teríamos diversos outros metadados que dependem da arquitetura do sistema que o algoritmo será implementado, como por exemplo o id de página e os ponteiros das listas encadeadas.

## 4.4 Fluxo de execução

A Figura 31 descreve o fluxo de execução do algoritmo EBRES. Semelhante aos algoritmos LRU-K e MQ, EBRES também usa um contador global *(GC)* que é incrementado para cada requisição ao *buffer*. Além disso, para cada requisição, é inserido uma entidade no histórico *in* com as informações do tipo da operação (leitura ou escrita) e se ocorreu um *hit* ou *miss* (ver Seção 4.5).

Figura 31 – EBRES fluxo de execução




Fonte: O Autor.

As novas páginas são inseridas nas listas *cold-clean* ou *cold-dirty*, dependendo do tipo da operação. Como discutido na Seção 4.3 há cinco metadados e eles são inicializados da seguinte forma: *references* recebe o valor 1, *last* recebe o valor atual do *GC*, *list* recebe um enumerador indicando em qual das listas *cold* a nova página reside, *SE* e *error* recebem o valor 0. Quando ocorre *hit* em alguma das quatro listas principais temos três casos:

- Requisições de leitura e *hits* na *cold-clean* ou *hot-clean* as páginas são movidas para a MRU da *hot-clean*.
- Requisições de leitura e *hits* na *cold-dirty* ou *hot-dirty* as páginas são movidas para a MRU da *hot-dirty*.
- Requisições de escrita e *hits* em qualquer uma das quatro listas principais as páginas são movidas para a MRU da *hot-dirty*.

Sempre que ocorre um *hit* em uma página o valor do campo *references* é incrementado em 1 e o enumerado do campo *list* é atualizado para a lista atual em que a página reside. Em seguida atualizamos os valores dos campos *SE*, *error* e *last* usando o Algoritmo 1. Onde $\alpha$ se refere ao fator de suavização.

**Algoritmo 1:** ES(P)

**Data:** Uma requisição para calcular o *SE* de uma página p

1 $distance \leftarrow GC - p.last$;
2 **if** $p.SE$ ***is 0*** **then**
3 $\quad p.SE \leftarrow distance$;
4 **else**
5 $\quad p.error \leftarrow \mid distance - p.SE \mid$;
6 **end**
7 $p.SE \leftarrow p.SE + \alpha * (distance - p.SE)$;
8 $p.last \leftarrow GS$;

Quando ocorre um *hit* no histórico *OUT*, primeiramente é preciso escolher uma vítima (ver Seção 4.6). Posteriormente, a estratégia recupera os metadados salvos dessa página *ghost* e carrega os dados da página na memória, assim como ARC e outras abordagens semelhantes. Além disso, também é executado o Algoritmo 1 para essa página.

## 4.5 Coletando estatísticas com o histórico *in*

O histórico *in* mantém informações de todas as últimas *IN_SIZE* requisições feitas ao *buffer*. Sempre que ocorre um acesso ao *buffer* independente se foi um *hit* ou *miss* uma nova

entidade é inserida. A estrutura funciona como um fila, caso o tamanho máximo *IN_SIZE* do histórico seja atingido, a entidade mais antiga (LRU) é descartada. Cada entidade do histórico mantém duas *flags*: *is_miss* e *is_dirty*, que informam se o histórico dessa requisição tenha sofrido um *hit* ou *miss* e se foi uma operação de leitura ou escrita. A ideia é que ao monitorar as últimas *IN_SIZE* requisições podemos extrair quatro estimativas:

- $read\_frequency$: Informa a porcentagem de requisições de leitura das últimas *IN_SIZE* requisições.
- $write\_frequency$: Informa a porcentagem de requisições de escrita das últimas *IN_SIZE* requisições.
- $read\_miss\_frequency$: Informa a porcentagem de requisições de leitura que tiveram um *miss* das últimas *IN_SIZE* requisições.
- $write\_miss\_frequency$: Informa a porcentagem de requisições de escrita que tiveram um *miss* das últimas *IN_SIZE* requisições.

Como mostra o Algoritmo 2, o processo para calcular as estimativas é bem simples, basta inicializar os valores das estimativas com 0 (linhas 1 a 5) em seguida percorrer as entradas do histórico *in* e realizar uma contagem das ocorrências (linhas 7 a 21) e por fim atualizar os valores das quatro estimativas de frequência (linhas 22 a 25).

A execução do procedimento de análise para cada requisição aumentaria significadamente a complexidade do algoritmo. Já que é preciso percorrer *IN_SIZE* entidades a cada chamada e torna o procedimento inviável. Para evitar esse problema, EBRES executa o procedimento de análise assincronamente ao gerenciador de *buffer*, mantendo uma complexidade constante durante o fluxo de execução. Ao invés de executar o procedimento para cada requisição, basta executa-lo em intervalos de tempo ou de requisições.

Outro fator determinante na performance do EBRES é a escolha do tamanho do *IN_SIZE*. Se o tamanho é muito grande, podemos está incluindo no cálculo algumas requisições muito antigas que podem não ter mais valor durante o contexto atual de execução. O contrário ocorre quando o tamanho é muito pequeno, podemos está perdendo informações importantes que podem refletir melhor o contexto de execução. Logo, é necessário um ajuste fino que mantenha um equilíbrio nesse aspecto.

**Algoritmo 2:** analyze()

```
   Data: Uma requisição para analisar o histórico IN
 1 counter ← 0;
 2 read_counter ← 0;
 3 write_counter ← 0;
 4 read_miss_counter ← 0;
 5 write_miss_counter ← 0;
 6 x ← IN.head;
 7 while x ≠ null do
 8     if x.is_dirty is true then
 9         write_counter ++;
10         if x.is_miss is true then
11             write_miss_counter ++;
12         end
13     else
14         read_counter ++;
15         if x.is_miss is true then
16             read_miss_counter ++;
17         end
18     end
19     x ← x.next;
20     counter ++;
21 end
22 read_frequency ← read_counter ÷ counter;
23 write_frequency ← write_counter ÷ counter;
24 read_miss_frequency ← read_miss_counter ÷ counter;
25 write_miss_frequency ← write_miss_counter ÷ counter;
```

## 4.6 Escolha da vítima

Algoritmos como o CASA, LLRU, e AM-LRU usam os parâmetros de ajuste para determinar o custo das operações de leitura e escrita. É possível escolher esses custos usando as informações da velocidade de acesso da própria mídia assimétrica ou de maneira mais subjetiva sugerindo ao algoritmo que o custo de escrita é maior que o de leitura. EBRES também faz uso deste recurso e utiliza os parâmetros *write_cost* e *read_cost*, os dois variam entre 0 a 1.

O Algoritmo 3 mostra como é feito o cálculo do custo de uma página *p*. Primeiro é calculado o *page_score* (linha 1) combinando a predição da suavização exponencial e o seu erro decomposto pelo o número de requisições. Logo, quanto maior o número de referências da página, menor será o custo. Porém, quando a página para de receber acessos, seu *p.SE* e *p.ERROR* aumentam e consequentemente aumentam o custo, indicando que essa página deixou de ser frequente.

**Algoritmo 3:** cost(p)

**Data:** Uma requisição para calcular o custo da página p
**Result:** Retorna o valor do custo da página p

```
1 page_score ← (p.SE + p.error) ÷ p.references;
2 if p.is_dirty is true then
3 |   return (page_score * write_frequency) + (1 − write_cost)
4 else
5 |   return (page_score * read_frequency) + (1 − read_cost)
6 end
```

O Algoritmo 4 descreve o processo de escolha da vítima. Primeiramente (linhas 1 a 4), selecionamos as quatro páginas LRU candidatas a vítima de cada uma das listas principais e guardamos suas referências no *array* $lru\_pages[\,]$ (linha 5), posteriormente usado para escolher uma das quatro candidatas a vítima.

Em seguida, calculamos o tamanho desejável das quatro listas (linhas 6 a 9) usando os parâmetros de ajuste dos tamanhos mínimos e as estatísticas de frequência das operações obtidas pelo histórico *IN*. Quando o número de páginas das listas tem o seu tamanho menor ou igual ao tamanho desejável, EBRES evita que as páginas dessa lista sejam substituídas (linhas 10 a 21), bastando, para isso, atribuir nulo na posição do *array* das LRU correspondentes.

Temos dois casos especiais (linhas 22 e 27), com a intenção de proteger as páginas que residem as listas *hot* de acessos sequenciais. EBRES usa as frequências de erro *(miss)*, o custo das operações e a capacidade do *buffer* ($BUFFER\_SIZE$) para identificar esse padrão de acesso, removendo imediatamente as páginas LRU de uma das listas *cold*.

Por fim, o algoritmo escolhe uma das quatro candidatas a vítimas guardadas no *array* (linha 32 a 39). Para cada candidata diferente de nulo, iremos calcular o custo da página usando o Algoritmo 3. Finalmente o algoritmo escolhe a página com o maior custo como a vítima. As páginas escolhidas como vítimas são despejadas do *buffer* e modificadas em uma página *ghost*, para serem inseridas no histórico *OUT*. Caso o tamanho do histórico *OUT* chegou no *OUT_SIZE*, a sua entidade mais antiga (LRU) é descartada da lista.

## 4.7 Análise do EBRES

Semelhante a Seção 3.4, apresentamos uma análise no algoritmo EBRES. Com as seguintes características:

- *Scan Resistance* (presente): EBRES é capaz de proteger as suas páginas frequentes quando

**Algoritmo 4:** get_victim()

**Data:** Uma requisição para escolher uma vítima
**Result:** Retorna a página vítima (*victim*)

```
1  P_CC ← cold-clean.LRU;
2  P_CD ← cold-dirty.LRU;
3  P_HC ← hot-clean.LRU;
4  P_HD ← hot-dirty.LRU;
5  lru_pages[ ] ← {P_CC, P_CD, P_HC, P_HD} ;
6  desirable_cold_clean = MIN_COLD_CLEAN * read_frequency;
7  desirable_cold_dirty = MIN_COLD_DIRTY * write_frequency;
8  desirable_hot_clean = MIN_HOT_CLEAN * read_frequency;
9  desirable_hot_dirty = MIN_HOT_DIRTY * write_frequency;
10 if cold-clean.size ≤ desirable_cold_clean then
11 |  lru_pages[0] ← null
12 end
13 if cold-dirty.size ≤ desirable_cold_dirty then
14 |  lru_pages[1] ← null
15 end
16 if hot-clean.size ≤ desirable_hot_clean then
17 |  lru_pages[2] ← null
18 end
19 if hot-dirty.size ≤ desirable_hot_dirty then
20 |  lru_pages[3] ← null
21 end
22 if read_miss_frequency > 0 then
23 |  if cold-clean.size > BUFFER_SIZE * read_miss_frequency * read_cost then
24 |  |  return P_CC
25 |  end
26 end
27 if write_miss_frequency > 0 then
28 |  if cold-dirty.size > BUFFER_SIZE * write_miss_frequency * write_cost then
29 |  |  return P_CD
30 |  end
31 end
32 victim ← null
33 for i = 0; i < 4; i++ do
34 |  if lru_pages[i] ≠ null then
35 |  |  if victim is null or cost(lru_pages[i]) > cost(victim) then
36 |  |  |  victim ← lru_pages[i]
37 |  |  end
38 |  end
39 end
40 return victim
```

opta em escolher a remoção de páginas das listas *cold* ao detectar por meio de estatísticas de frequência, um possível acesso sequencial.

- *Low Lifetime Page Protection* (presente): Os tamanhos mínimos de cada lista mitigam que as novas páginas tenham um curto período de vida. Mesmo em um cenário extremo com uma carga de trabalho totalmente de apenas uma operação. A suavização exponencial e o erro das novas páginas são inicializados com 0, fazendo que o custo dessa página seja baixo, e assim evitando uma possível remoção imediata, já que as páginas frequentes terão seu *page score* maior que 0 e consequentemente um custo maior que uma nova página.
- *Page Frequency Levels* (presente): EBRES utiliza um contador de acessos (*references*) assim tendo a característica de diferentes níveis de frequência.
- *Aging Mechanism* (presente): Como as quatro listas tem tamanhos dinâmicos, a medida que a carga de trabalho é executada as páginas envelhecem até chegarem na posição LRU da sua lista e assim sendo escolhidas como vítima.
- *Tuning Parameters* (presente): EBRES apresenta diversos parâmetros de ajuste: O fator da suavização exponencial $\alpha$, tamanhos dos históricos $IN\_SIZE$ e $OUT\_SIZE$, os tamanhos mínimos de cada uma das quatro listas principais e por fim, os custos das operações $read\_cost$ e $write\_cost$.
- *History Mechanism* (presente): EBRES utiliza dois mecanismos de histórico. *IN* é responsável por coletar estatísticas da carga de trabalho armazenado metadados sobre as requisições. *OUT* é responsável por armazenar os dados das páginas escolhidas como vítima. Logo, se uma página vítima é acessada novamente e seus metadados residem no histórico *OUT* podemos tirar proveito das informações antigas, como por exemplo, o valor da sua última predição do modelo de suavização exponencial.
- *Constant Complexity* (presente): O mecanismo de seleção de vítimas e a execução do modelo de suavização exponencial apresentam complexidade computacional constante.
- *Cache Starvation Issues* (não presente): EBRES não apresenta esse problema, pois mesmo em um cenário extremo que força alguma lista a ficar vazia, o mecanismo da estratégia permite que ela cresça novamente com a mudança da carga de trabalho.
- *Clustered Writes* (não presente): Os mecanismos de agrupamento de páginas de escrita propostos pelas estratégias CFDC e SCMBP-SCCW exigem um esforço adicional pois precisam manter todo o mecanismo de agrupamento, aumentando a complexidade do algoritmo principalmente em uma carga de trabalho majoritariamente de escrita. Logo, para

manter uma complexidade constante nos não optamos em implementar esse mecanismo ainda. Todavia, podemos adaptar o EBRES usando o CFDC como base, dividindo o *buffer* com uma parte da memória dedicada o EBRES e a outra em manter as páginas vítimas de escrita do EBRES em um mecanismo de agrupamento, para posteriormente serem despejadas.

## 4.8 Conclusão

Este capítulo descreve a estratégia de substituição de páginas proposta denominada EBRES. Usando o modelo de suavização exponencial e mecanismos de ajuste baseados em estatísticas da carga de trabalho, em conjunto com quatro listas principais e duas listas auxiliares, EBRES busca um manter equilíbrio com diferentes cargas de trabalho, reduzindo operações de escrita e melhorando a taxa de acerto com uma complexidade constante. Por fim, a Seção 4.7 apresenta uma análise do EBRES usando os mesmos critérios da Seção 3.4.

# 5 AVALIAÇÃO EXPERIMENTAL

Neste capítulo, apresentamos os experimentos que foram realizados e os resultados obtidos. Para isso, este capítulo está assim dividido: a Seção 5.1 descreve o ambiente das experimentações, destacando os *benchmarks*, implementação do gerenciador de *buffer* e as configurações da máquina de teste; as Seções 5.2 e 5.4 apresentam os resultados da avaliação junto com as discussões a partir destes resultados.

## 5.1 Configuração do Ambiente de Experimentação

Para avaliar o EBRES utilizamos três *traces* reais: TPC-C (ASSOCIATION, 2007b), TPC-E (ASSOCIATION, 2007c) e File Server (ASSOCIATION, 2007a); e dois *traces* sintéticos, chamados de ZIPF (TAVARES, 2015) que foram gerados com 20% e 80% de operações de escrita.

Tabela 3 – Descrição dos *traces* usados

| Nome | Requisições | Leitura | Escrita | Páginas Acessadas | Tamanho do BD em páginas |
|---|---|---|---|---|---|
| File Server | 1.175.607 | 790.070 | 385.537 | 487.495 (1,9 GB) | 2.778.458 (11,3 GB) |
| TPC-C | 3.000.000 | 2.811.727 | 188.273 | 919.367 (3,7 GB) | 6.929.239 (28,3 GB) |
| TPC-E | 3.000.000 | 2.864.708 | 135.292 | 1.482.576 (6,0 GB) | 5.264.225 (21,5 GB) |
| ZIPF20%W | 1.500.000 | 1.200.000 | 300.000 | 285.281 (1,1 GB) | 999.993 (4,0 GB) |
| ZIPF80%W | 1.500.000 | 300.000 | 1.200.000 | 285.281 (1,1 GB) | 999.993 (4,0 GB) |

Fonte: O Autor.

A Tabela 3 apresenta as principais características de cada *trace*. O número de requisições representa a quantidade total de operações (leitura ou escrita) que acessaram páginas do banco de dados. Do número total requisições, separou-se a quantidade de acesso para leitura e a quantidade para escrita. As páginas acessadas informam a quantidade de dados acessados do tamanho total do banco de dados do *trace*. E por fim, o tamanho do BD em páginas corresponde ao tamanho da base de dados completa em páginas. Para garantir uma experimentação correta, foi definido que as páginas dos bancos de dados têm tamanho de 4kB.

O *trace* File Server apresenta requisições do Microsoft Storage, uma aplicação que possibilita o compartilhamento e acesso simultâneo de arquivos entre usuários de uma rede de computadores. Os *traces* TPC-C e TPC-E são cargas de trabalho com o foco no processamento de transações e são referência em avaliações em diversos trabalhos que envolvem operações transacionais. Estes *traces* apresentam uma intensidade de operações de leitura significativa,

embora tenham o mesmo número de requisições, o TPC-C apresenta um número de páginas acessadas menor que o TPC-E. Como o conjunto de páginas acessadas é menor (quase 1/3 do número de requisições), a ocorrência de páginas frequentes será maior, dado o número de requisições da carga de trabalho. Por fim, os *traces* ZIPF exploram umas cargas de trabalho extrema, com uma forte intensidade de operações de leitura ou de escrita (TAVARES, 2015).

Foi implementado um gerenciador de arquivos e gerenciador de *buffer* [1] em linguagem C, capaz de realizar chamadas diretas ao sistema operacional e emular o processamento das transações usando os *traces*. O gerenciador de *buffer* encapsula diferentes políticas de substituição de páginas, onde em cada experimento apenas uma é ativada. Para comparação, usamos os algoritmos LRU, ARC e LIRS, CFDC e AM-LRU (ver Seções 2.2, 3.1.9, 3.1.8, 3.2.4, respectivamente). O LRU é um algoritmo base de fácil implementação e é frequentemente usado como comparativo em diversos trabalhos. ARC e LIRS são conhecidos pelos seus mecanismos que tratam a frequência ou recência de páginas. Já o CFDC e AM-LRU têm foco em considerar também a assimetria da mídia de armazenamento, com relação a operações de leitura e de escrita. Os experimentos foram executados em uma máquina de 64 GB de RAM e 480 GB de SSD Kingston SA400S37480G com Ubuntu Linux 18.04 LTS Kernel 4.15.0-74-generic, Intel i7-9700k 3,60 GHz.

## 5.2 Configurações Gerais dos Algoritmos Testados

Para todos os algoritmos, foram executados os *traces*, variando o tamanho do *buffer* (em páginas) em 0,1%, 1%, 2%, 5% e 10% da quantidade das páginas acessadas (eixo x, nos gráficos). Computamos o tempo de execução em segundos, o número de *hits* e escritas (eixo y, nos gráficos). Como já mencionado, o gerenciador de arquivos usou o tamanho do bloco de dados de 4kB. Como parâmetro de ajuste, no CFDC foi usado 50% do tamanho do *buffer* na *working region*. Para o LIRS, 1% do tamanho do *buffer* para páginas do tipo HIR. Para o AM-LRU, o *min_len* foi usado 10% do tamanho do *buffer*. Para o EBRES, o *IN_SIZE* foi configurado como 10% do tamanho do *buffer*, *OUT_SIZE* igual ao tamanho do *buffer*, os tamanhos mínimos das quatro listas principais como 10% do tamanho do *buffer*, e os custos *read_cost* e *write_cost*, 0,2 e 0,8. Por fim, o $\alpha$ como 0,5.

Diversos fatores podem influenciar o tempo de execução dos experimentos (ver Figuras 34, 37, 40, 43, 46): fatores externos, como o sistema operacional e procedimentos

[1] O código dos gerenciadores implementados pode ser acessados em github.com/ggustavo/SNK-DB

internos do SSD podem contribuir com variações de tempo, e internos, como operações de escritas e de leitura (*miss*), fazem com que o tempo de execução aumente, pois o sistema realiza acessos diretos a mídia de armazenamento. Outro fator é a complexidade dos algoritmos, por exemplo o CFDC em vários cenários apresenta um tempo de execução elevado em comparação aos outros algoritmos. Acessos randômicos, podem prejudicar seu mecanismo de *clusters*, fazendo com que o CFDC precise manter a ordem das prioridades de vários *clusters* de apenas uma página. No LIRS, a operação de *stack pruning* e no AM-LRU, o mecanismo de segunda chance para páginas frequentes, quando executados sucessivas vezes, podem impactar diretamente o tempo de execução. Já as estratégias ARC, LRU, e EBRES como possuem uma complexidade constante, seu principal impacto no tempo de execução são as operações internas de leitura e escrita e os fatores externos.

## 5.3 Análise das Cargas de Trabalho

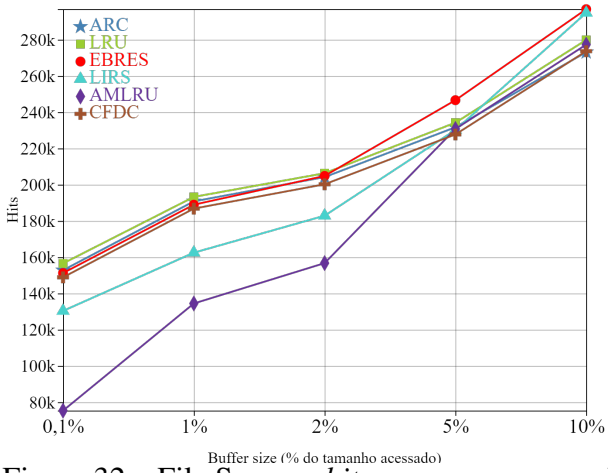



Figura 32 – File Server - *hits*

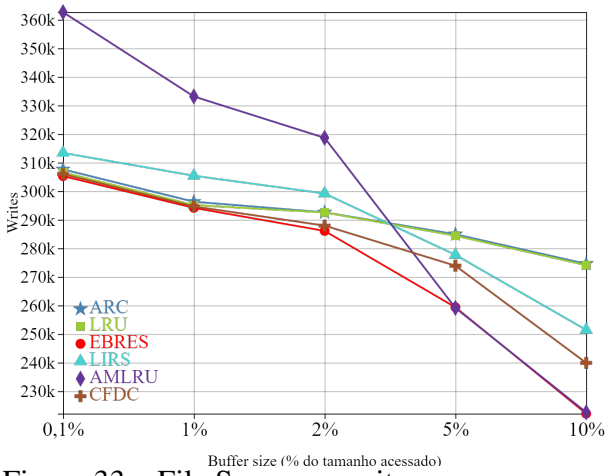



Figura 33 – File Server - escritas

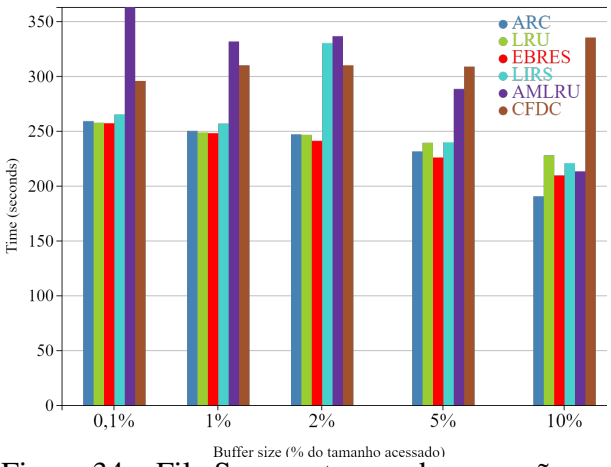



Figura 34 – File Server - tempo de execução

Para o File Server (Figuras 32, 33 e 34), temos uma carga de trabalho real de 67,21% de leituras e 32,79% de escritas. LIRS e AM-LRU possuem políticas mais sensíveis para páginas recentes e é possível que isso possa impactar os *hits* e escritas para tamanhos de *buffer* menores entre 0,1% e 2%, já que essas páginas podem precisar de um tempo de vida maior no *buffer* para serem acessadas novamente e consideradas frequentes. Para os demais algoritmos, nos tamanhos menores 0,1% e 2%, apresentam um número de *hits* equivalentes. EBRES, para todos os tamanhos, obteve um desempenho equilibrado de *hits* e número baixo de escritas, todavia no tamanho de 10% obteve o melhor desempenho nestes dois quesitos. No tamanho de 0,1% fica perceptível que o AM-LRU apresentou um elevado tempo de execução, principalmente por conta do seu baixo número de *hits* e número elevado de escritas. Algo semelhante ocorre com o CFDC, porém é provável que a causa seja a degradação do mecanismo de clusterização, que tende a piorar em tamanhos de *buffer* maiores, já que o número de *clusters* tende aumentar e consequentemente o custo da ordenação também.

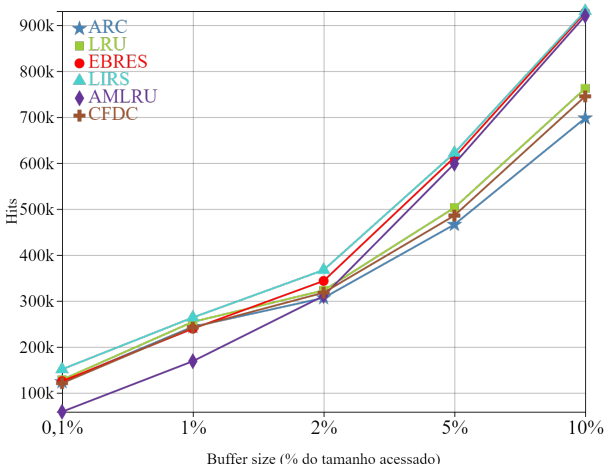



Figura 35 – TPC-C - *hits*

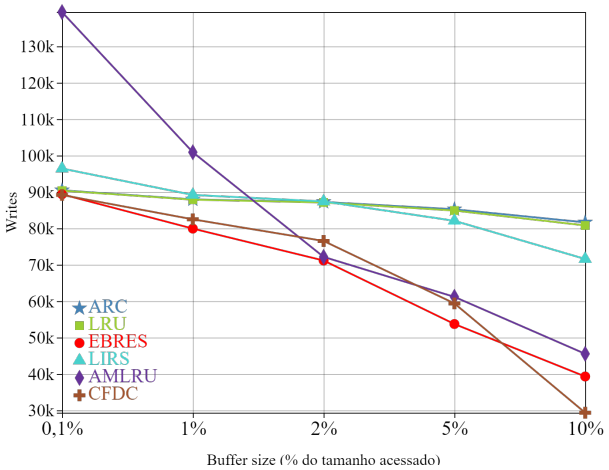



Figura 36 – TPC-C - escritas

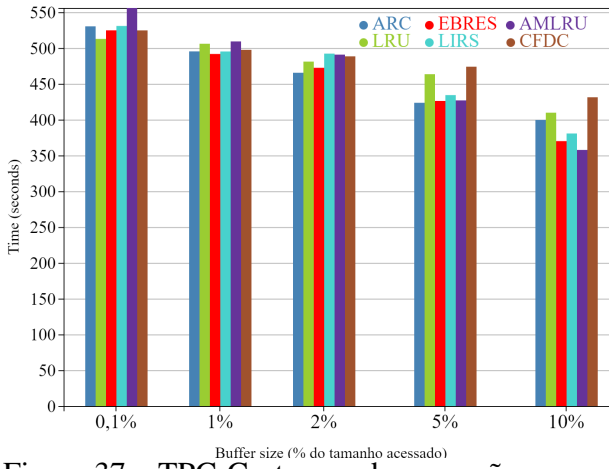



Figura 37 – TPC-C - tempo de execução

Para o TPC-C (Figuras 35, 36 e 37), temos uma carga de trabalho majoritariamente de leitura (93,72%). No número de *hits*, EBRES obteve um desempenho parcialmente inferior ao LIRS em todos os tamanhos, em especial nos tamanhos de *buffer* entre 0,1% e 1%, entretanto, a diferença média ficou em torno de 7%. Por outro lado, EBRES obteve o menor número de escritas em todos os tamanhos, exceto nos tamanhos de 0,1% e 10%, em que o CFDC atinge um melhor resultado, variando em apenas 0,2% no tamanho 0,1% e 33,93% no tamanho de 10%. Para os algoritmos com complexidade constante ARC, LRU e EBRES, o tempo de execução diminui a medida que o tamanho do *buffer* aumenta.

Como discutido anteriormente, a carga de trabalho do TPC-C tende a ter um número maior de *hits* do que a carga TPC-E por consequência do seu número de páginas acessadas ser bem menor. No entanto, até que um *hit* ocorra o algoritmo precisa manter a página no seu *buffer* por um determinado período. Esse período, está principalmente relacionado com o tamanho do *buffer* em páginas e também quantas páginas o algoritmo deve manter na sua região de recência por exemplo. Na tentativa de prolongar um pouco o tempo de vida das páginas e tentar descobrir quais delas serão frequentes, algoritmos como ARC, LIRS e EBRES possui um histórico (páginas *ghosts*) com os metadados das requisições que foram despejadas do *buffer*, embora não consigam de fato um *hit* real. Todavia, ao localizar uma página no histórico, podemos detectar que ela deveria estar no *buffer* e ter tido um tempo de vida maior, cabe então o algoritmo usar essa informação ao seu favor e conseguir se adaptar.

Durante o experimento do TPC-C, ARC apresentou dificuldades no número de *hits*, tendo um desempenho menor que o LRU, intensificado um pouco mais nos tamanhos de 5% e 10%. Como ARC divide as suas páginas nas regiões de recência e frequência e os metadados das suas vítimas são enviadas para os históricos também de recência e frequência, ele tende a ter diferentes tempos de vida para páginas dependendo da carga de trabalho. Contudo, durante possíveis operações sequenciais (melhor discutido na Seção 5.4) em que o ARC se adapta e tende remover páginas recentes primeiro, em um caso específico ele não envia essas páginas vítimas para o seu histórico de recência. É comum que nas requisições de uma carga de trabalho incluam páginas candidatas a serem frequentes no meio de operações sequenciais, uma vez que as operações das transações do banco de dados são geralmente executadas concorrentemente entrelaçadas (HELLERSTEIN *et al.*, 2007). Como o ARC acaba deixando de armazenar páginas no seu histórico isso pode ter influenciado no seu desempenho.

Para o TPC-E (Figuras 38, 39 e 40), EBRES apresenta um desempenho mediano

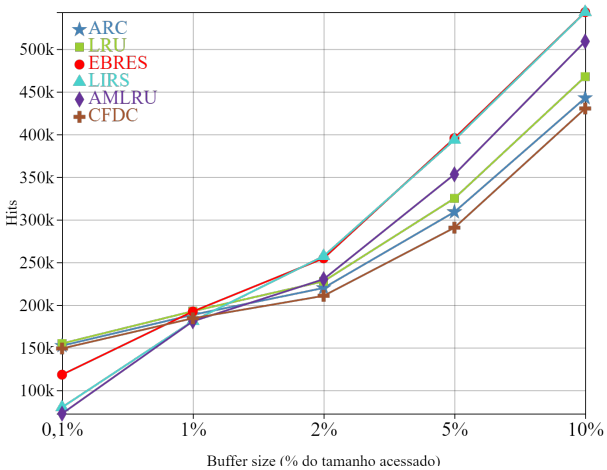



Figura 38 – TPC-E - *hits*

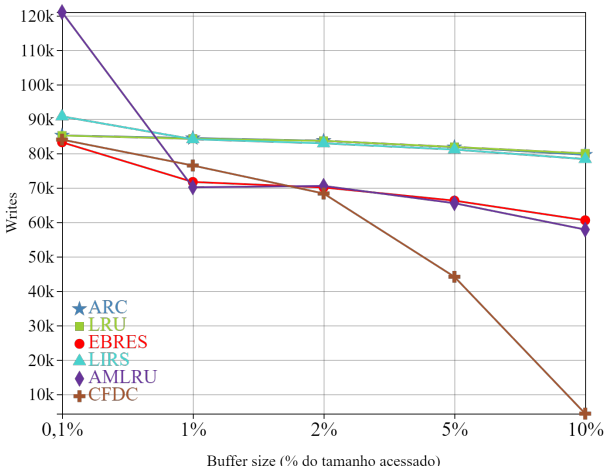



Figura 39 – TPC-E - escritas

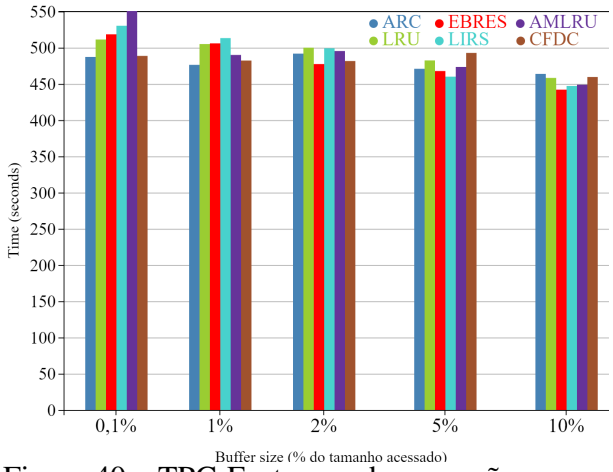



Figura 40 – TPC-E - tempo de execução

comparado com os demais algoritmos no tamanho de 0,1%, no número de *hits*. Entre os tamanhos de 1% e 10% EBRES mantém um desempenho semelhante ao LIRS e AM-LRU, em termos de *hits* e escritas, respectivamente. Embora o CFDC tenha obtido o menor número de escritas, para os tamanhos de *buffer* de 2% ou maior, com uma diferença significativa dos demais algoritmos no tamanho de 10%, ele teve o menor número de *hits*, com uma leve diferença do ARC. O problema do *cache starvation* (ver Seção 2.3.2) pode ter impactado para diminuir o número de escritas. Como a carga de trabalho é majoritariamente de leitura (95,49%), a região de leitura da *priority region* é incapaz de crescer quando a região de escrita cresce e toma seu espaço, o CFDC foi capaz de manter aproximadamente 50% do seu espaço do *buffer* apenas para páginas de escrita, reduzindo significativamente o número de escritas, isso intensifica ainda mais, caso essas páginas de escrita sejam frequentes e possam residir na *working region*.

Ainda no TPC-E, notamos que o LIRS apresentou dificuldades nos tamanhos pequenos de 0,1% e 1% no seu número de *hits*. Diferente do experimento anterior do TPC-C, o *trace* do TPC-E apresenta uma incidência menor de páginas frequentes, isso faz que o mecanismo de

histórico não tenha tanta influência, já que as mesmas páginas não são acessadas constantemente, por exemplo, casos em que uma página é acessada apenas duas ou três vezes em um curto período de tempo. Logo, temos uma influência maior do tempo de vida das páginas recentes, fazendo com que o LRU, que apresenta um elevado tempo de vida (discutido melhor na Seção 5.4), tenha um bom desempenho nestes tamanhos de *buffer* pequenos. Semelhante ao LRU, o ARC e CFDC também se destacaram no tamanho 1%. No caso do ARC, com um número menor de páginas frequentes, teríamos uma região de recência maior, consequentemente um tempo de vida maior. No caso do CFDC, pelo menos 50% do seu *buffer* (*working region*) é usado para manter páginas recentes até serem envelhecidas e movidas para a sua *priority region*. A partir que o tamanho do *buffer* aumenta (2% em diante), os demais algoritmos conseguem ter um tamanho maior para as suas regiões de recência, de maneira que agora os seus históricos e o cuidado com as páginas frequentes tenham um maior impacto.

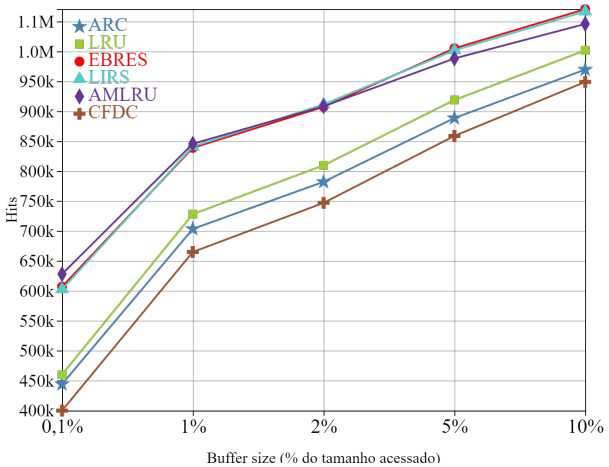



Figura 41 – ZIPF 20%W - *hits*

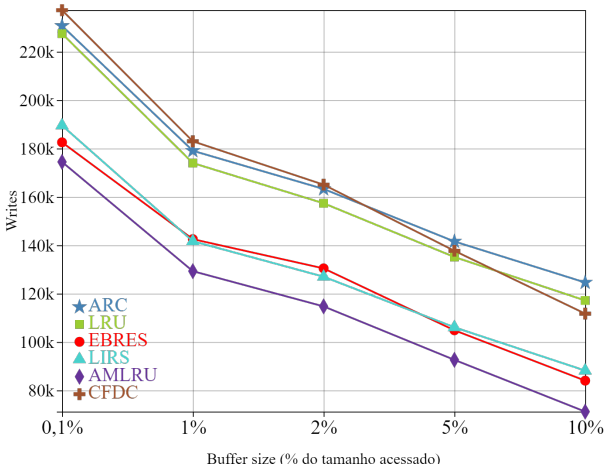



Figura 42 – ZIPF 20%W - escritas

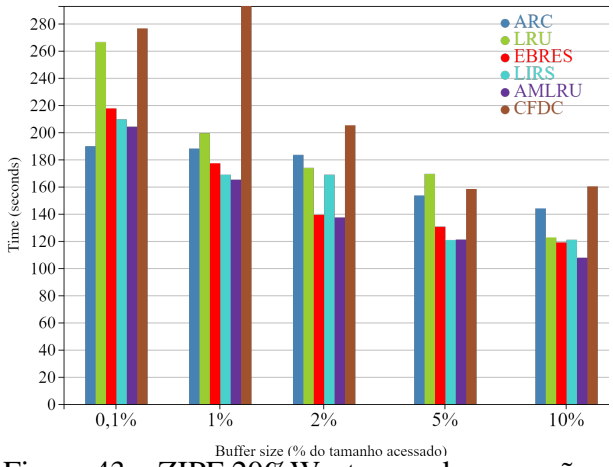



Figura 43 – ZIPF 20%W - tempo de execução

Para o ZIPF 20%W (Figuras 41, 42 e 43), para todos os tamanhos de *buffer*, AM-

LRU apresentou em média 11,95% menos escritas que o EBRES e uma variação média de 1,69% nos *hits* em relação ao EBRES. Seu tempo de execução também foi decrescente a medida que o tamanho do *buffer* aumenta. Parte das páginas classificadas como frequentes pelo LIRS também são páginas de escritas, mesmo a estratégia não considerando o tipo de operação, ela tende a ter bons resultados. Um dos impactos negativos do CFDC, durante os testes é devido a sua prioridade elevada em manter várias páginas de escrita no *buffer*, mesmo que hipoteticamente não sejam frequentes.

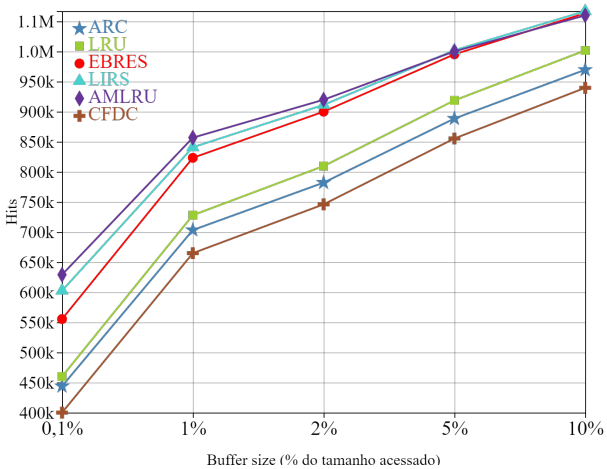



Figura 44 – ZIPF 80%W - *hits*

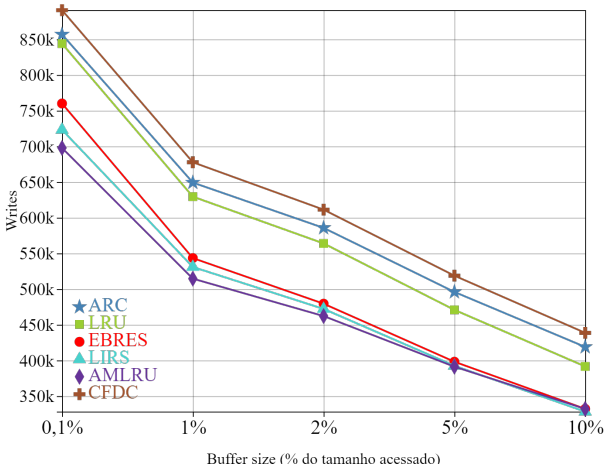



Figura 45 – ZIPF 80%W - escritas

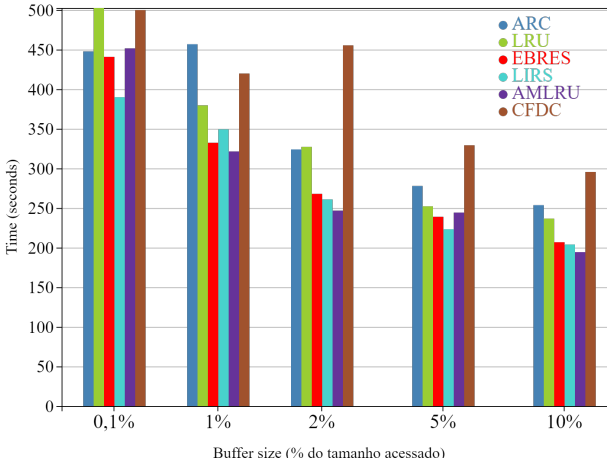



Figura 46 – ZIPF 80%W - tempo de execução

Os dois *traces* sintéticos ZIPF de acessos randômicos apresentam a mesma ordem de requisições variando apenas o tipo da operação. Logo, nota-se que temos um comportamento bem similar em relação ao visual dos gráficos, porém diferem nos resultados, por exemplo o número de escritas. Para o *trace* ZIPF 80%W (Figuras 44, 45 e 46), os algoritmos LRU, ARC e CFDC apresentam um menor rendimento do que os algoritmos AM-LRU, LIRS e EBRES. Para esta carga com maioria de operações de escrita, temos uma influência maior dessas operações

no tempo de execução. Algoritmos que realizaram mais escritas, tendem apresentar um tempo de execução um pouco maior. AM-LRU, LIRS e EBRES a partir do tamanho *buffer* de 2% apresentam resultados semelhantes.

Na questão de assimetria, é esperado que os algoritmos EBRES, AM-LRU e CFDC que consideram essa característica tenham um número de escritas reduzido. Nos testes do File Server, TPC-C, e TPC-E (Figuras 33, 36, 39, respectivamente) notamos que o algoritmo AM-LRU apresenta um número elevado de escritas em tamanhos pequenos de *buffer*; porém, percebemos que ao mesmo tempo o AM-LRU também tem um número de *hits* baixo (tamanhos de 0,1%), por consequência também um tempo de execução maior, principalmente no teste do File Server. Um dos motivos para esse baixo desempenho é devido as características do algoritmo serem sensíveis a recência, podendo apresentar um baixo tempo de vida para as páginas e como o AM-LRU não possui um histórico (páginas *ghosts*) igual ao LIRS, isso dificulta a identificação de páginas frequentes. O algoritmo pode apresentar dificuldades para envelhecer as páginas frequentes, pois seus mecanismos tendem a priorizar a remoção de páginas recentes.

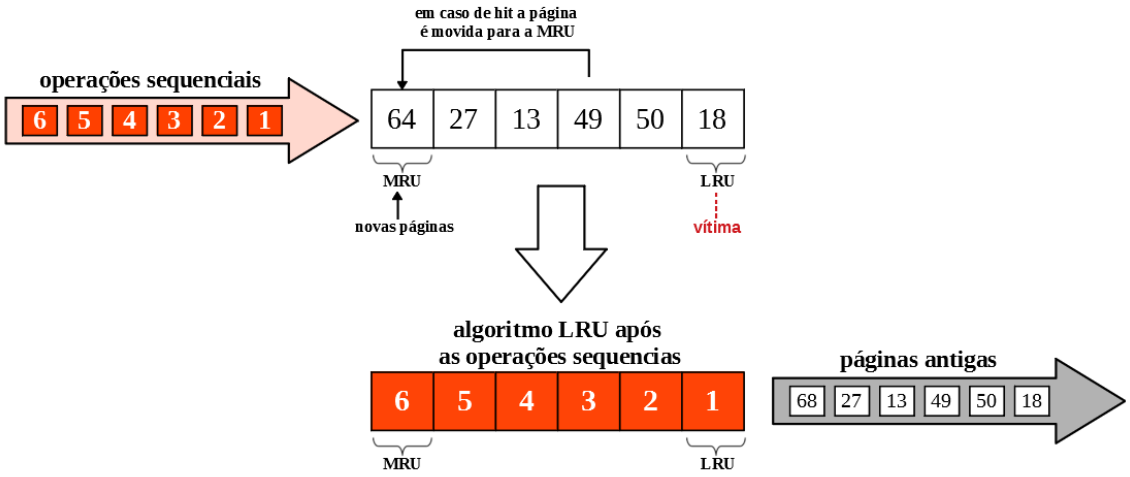
Ambas as complicações do AM-LRU relacionadas ao tempo de vida e envelhecimento das páginas frequentes com tamanhos pequenos de *buffer* são minimizadas nos dois experimentos sintéticos dos ZIPF (Figuras 42 e 45). Primeiramente, devemos observar que essas duas cargas de trabalhos apresentam um número muito baixo de páginas acessadas (quase 1/5 do número de requisições). Dessa forma, tendem a ter uma maior incidência de páginas frequentes expressivamente maior que o TPC-C e os outros experimentos. Como a probabilidade uma página já classificada frequentes pelo AM-LRU ser referenciada no futuro é grande, os problemas com envelhecimento são contidos. De certa forma, para essas duas cargas de trabalho, se tornou vantajoso priorizar a remoção de páginas recentes.

Embora os experimentos sintéticos ZIPF sejam antagônicos em relação ao seu número de leituras e escritas, as suas requisições de páginas são as mesmas. Dessa forma, os algoritmos que melhor se adaptaram e identificaram as páginas frequentes devem realizar também menos escritas. É vantajoso quando uma página frequente de escrita reside no *buffer*, porque podemos está favorecendo os dois aspectos do número de *hits* e escritas, e consequentemente também o tempo de execução. Em síntese, em ambos os experimentos ZIPF, os algoritmos AM-LRU, LIRS e EBRES melhor se adaptaram na identificação de páginas frequentes, consequentemente foram capazes de reduzir o número de escritas e o tempo de execução.

## 5.4 Avaliação do Mecanismo de *Scan Resistance*

Esta seção tem como objetivo analisar os mecanismos de resistência a operações sequenciais do EBRES e dos demais algoritmos avaliados na Seção 5.2. Como dito na Seção 2.3.1, as operações sequenciais são comuns em cargas de trabalho durante o processamento de transações. Por exemplo, operações de leitura durante um *table scan* ou operações de escrita por longas inserções durante uma importação de dados. Nem sempre todas as páginas das operações sequenciais são utilizadas brevemente, por exemplo, no caso de um *table scan* disparado por uma consulta simples de seleção, as páginas com dados que não satisfazem a condição de busca possuem uma alta probabilidade de não serem referenciadas novamente a curto prazo.

Figura 47 – Operações sequenciais no algoritmo LRU

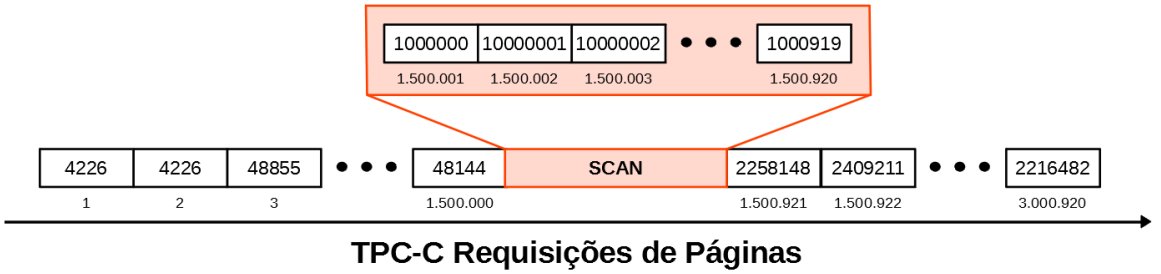



Fonte: O Autor.

O algoritmo LRU não apresenta nenhum mecanismo de *scan resistance*, a Figura 47 descreve o pior caso do algoritmo LRU que é provocado quando ocorre requisições de um *scan* com páginas distintas igual ou maior que o tamanho do *buffer* em páginas. A medida que as requisições chegam as páginas antigas são deslocadas em direção ao lado LRU da lista até serem totalmente substituídas. Algumas dessas páginas antigas potencialmente seriam ou são páginas frequentes e utilizadas por outras transações. Consequentemente, há um "envelhecimento"artificial das páginas, provocando uma queda na taxa de acerto (*hits*) do *buffer*.

Para realizar esses experimentos foram utilizadas as mesmas especificações descritas nas Seções 5.1 e 5.2. O *trace* escolhido foi o TPC-C pois se trata de uma carga de trabalho real com uma maior incidência de páginas frequentes em relação aos outros dois *traces* reais da Tabela 3.

Figura 48 – Inserindo uma operação sequencial no TPC-C




Fonte: O Autor.

Embora o *trace* do TPC-C contenha operações sequenciais distribuídas na sua carga de trabalho, não temos um pleno controle do número de requisições, início, fim e quais páginas serão requisitadas. Logo, para esses experimentos específicos, optamos em simular uma operação sequencial conforme apresentado na Figura 48, a ideia é que essa operação sequencial artificial seja um cenário extremo que não produz nenhum *hit* no *buffer* durante a sua execução e causa o pior caso do algoritmo LRU (ver Figura 47). Foi inserido no *trace* do TPC-C um *scan* com ID's de páginas que serão requisitados apenas uma vez durante toda a execução do *trace*, ou seja são páginas recentes que são excelentes candidatas a serem despejadas.

O *scan* artificial foi inserido no meio do *trace* do TPC-C, desta forma os algoritmos dispõe de um longo período para se adaptar a carga de trabalho e classificar páginas frequentes. Em cada teste foi variado o tamanho do *buffer* (em páginas) em 0,1%, 1%, 2%, 5% e 10% da quantidade das páginas acessadas do *trace*. Logo o número de requisições do *scan* em cada teste corresponde ao mesmo tamanho do *buffer*. Desta forma é possível provocar o pior caso da LRU, pois as requisições do *scan* devem inundar todo o seu *buffer*. Após o fim do *scan* podemos avaliar se os algoritmos são capazes de reagir ao padrão e evitarem que suas páginas classificadas como importantes sejam substituídas por páginas do *scan*.

As Figuras 49, 50, 51, 52, 53 e 54 avaliam os mecanismos de *scan resistance* dos algoritmos durante a execução de um *scan*. Em cada gráfico observamos um recorte da queda de desempenho provocado pelo *scan* no número de *hits* (eixo y) em cada tempo que as páginas são requisitadas (eixo x), sendo esse recorte mostrado um pouco antes da requisição 1.500.000 (meio do *trace*), durante e após o fim do *scan*. A linha pontilhada vertical laranja representa o início do *scan* e a linha cinza o seu fim. Durante o período de atuação do *scan* notamos que todos os algoritmos pararam de receber *hits*, gerando uma linha reta nos gráficos.

Nas Figuras 49 e 50 com *scans* maiores em *buffers* com os tamanhos de 10% (91.936) e 5% (45.968) da quantidade das páginas acessadas; em ambos os experimentos após o fim

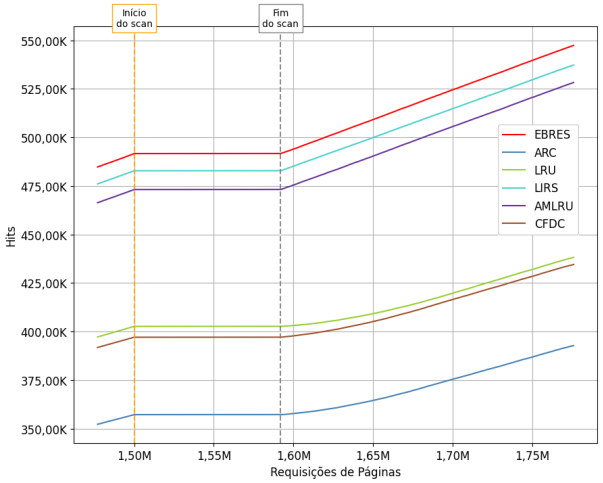



Figura 49 – 10%, *scan* de 91.936 páginas

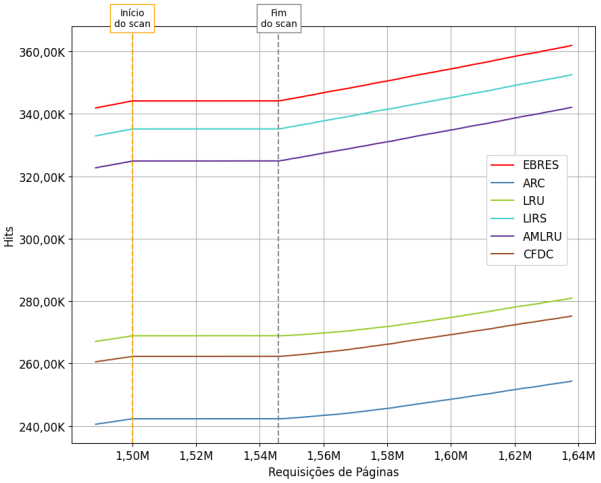



Figura 50 – 5%, *scan* de 45.968 páginas

do *scan* notamos que os algoritmos EBRES, LIRS e AM-LRU imediatamente retornam a ter *hits* fazendo que a inclinação da curva do seu gráfico seja expressivamente maior que os outros algoritmos avaliados. Isso significa que esses algoritmos conseguiram identificar o padrão e protegeram páginas importantes que seriam acessadas depois que *scan* fosse finalizado.

Os algoritmos LRU e CFDC não apresentam mecanismos de *scan resistance*, embora o ARC possua um mecanismo de *scan resistance*, nem sempre esse mecanismo é tão rigoroso quanto o do EBRES, LIRS e AM-LRU. Além disso, seu desempenho já vinha sendo inferior antes do início do *scan*; isso pode ser ocasionado pelas variações da carga de trabalho, dentre elas *scans* reais do próprio TPC-C que vão aos poucos degradando a performance dos algoritmos.

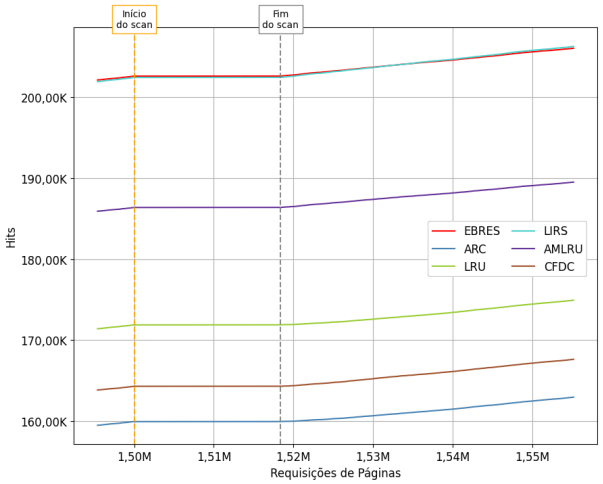



Figura 51 – 2%, *scan* de 18.388 páginas

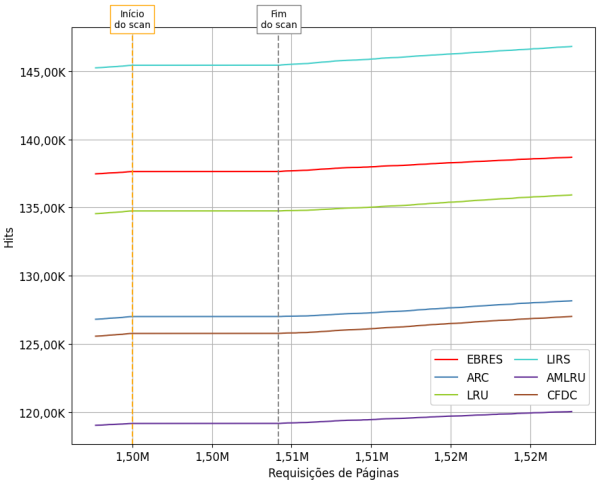



Figura 52 – 1%, *scan* de 9.194 páginas

Quando o tamanho do *buffer* é reduzido para 2% (18.388), percebemos na Figura 51 que a inclinação da curva de todos os algoritmos avaliados diminuiu após o fim do *scan* e todas as curvas são bem similares, porém levemente maior no EBRES, LIRS e AM-LRU demonstrando

que todos os algoritmos estão apresentando dificuldades. Notamos também que o EBRES e LIRS apresentam um desempenho muito similar neste tamanho de *buffer*.

Quando o tamanho do *buffer* é reduzido ainda mais para 1% (9.194) na Figura 52, fica claro que o LIRS apresentou a melhor tolerância ao *scan*, isso se dá devido ao seu mecanismo ser o mais severo entre os outros algoritmos avaliados no quesito de *scan resistance*. Porém esse mecanismo do LIRS tem um custo, uma vez que aumenta muito o consumo de memória enquanto armazena páginas de metadados (*ghosts*) das substituições feitas e também exige mais consumo de CPU para executar sua operação de *stack pruning*. Para esse tamanho de *buffer* menor, o ARC teve melhoras no desempenho, superando o CFDC, enquanto o AM-LRU teve uma queda. Isto será melhor evidenciado nos próximos parágrafos.

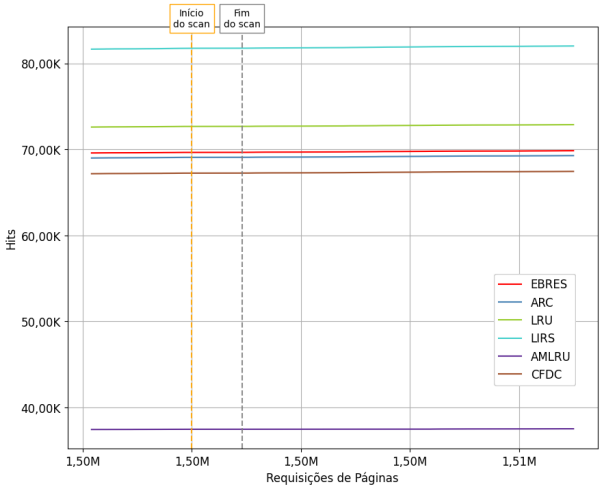



Figura 53 – 0,1%, *scan* de 920 páginas

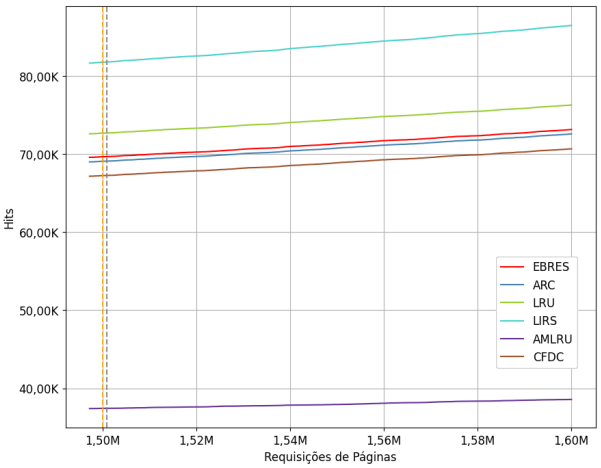



Figura 54 – 0,1%, *scan* de 920 páginas: análise da inclinação das curvas

Quando o tamanho do *buffer* é muito pequeno 0,1% (920), na Figura 53 observamos que mesmo sem o *scan* os algoritmos já estavam com muita dificuldade. Isso é um indicativo que independente do algoritmo essa carga de trabalho exige mais memória. Neste cenário extremo com o TPC-C quando calculamos o consumo de memória das páginas temos um banco de dados de 28,3 GB com uma carga de trabalho de páginas acessadas de 3,7 GB sendo executado com um *buffer* de apenas 3,9 MB. Após o fim do *scan*, os algoritmos recebem *hits*, porém as inclinações das curvas são sutilmente notáveis no gráfico.

Ainda no mesmo teste da Figura 53, para uma melhor análise da inclinação das curvas, a Figura 54 aumenta a perspectiva ao visualizar mais requisições após o fim do *scan*, embora o *scan* agora aparente não ter nenhum efeito negativo, em um cenário de uma carga de trabalho analítica como do TPC-C com vários *scans*, aos poucos a performance do *buffer* é

degradada[2].

Neste cenário extremo, quando uma página específica é requisitada é possível que a distância entre a sua penúltima requisição seja muito grande, pois o tamanho do *buffer* reduz a chance desta página já residir na memória antes da sua próxima requisição (*hit*). Por exemplo, quando uma nova página chega ao algoritmo LRU, o seu tempo de vida no *buffer*, caso não receba nenhum *hit*, é igual ao tamanho do *buffer* - 1 em requisições de *miss*. Pois essa página deve ser deslocada até o final da lista. Logo, se a distância entre as requisições de uma mesma página ultrapassa esse tempo de vida, o algoritmo pode deixar de classificar a página como frequente. Como discutido previamente, esse problema pode ser mitigado com o uso de páginas *ghosts*. Dado que quando a página é requisitada novamente e uma *ghost* com o mesmo ID já reside no *buffer*, ela entrega a informação que essa página pode ser considerada frequente.

O LIRS apresenta o menor tempo de vida para as páginas recentes e ainda assim apresentou um ótimo desempenho em tamanhos de *buffer* menores. O motivo é que ele é capaz de armazenar um número vasto e indeterminado de páginas *ghost* e assim reconhece as páginas frequentes. Porém, como dito anteriormente, o seu consumo de memória é alto.

Embora os algoritmos EBRES, AM-LRU e ARC adaptem o tamanho das suas regiões de páginas recentes, para essa carga de trabalho com páginas frequentes, dificilmente terão um tempo de vida maior que o LRU, pois parte do *buffer* destes algoritmos é destinado exclusivamente para páginas frequentes.

Por conta do tempo de vida maior, o LRU acaba tendo vantagem quando temos um cenário mais extremo, com tamanhos *buffer* pequenos, sendo superado apenas pelo LIRS. Pelo menos 50% do *buffer* do CFDC se comporta semelhante ao LRU (*working region*), por conta do problema do *cache starvation*, a outra parte do seu *buffer* (*priority region*) pode apresentar um número alto de páginas *dirty* fazendo com que essas páginas tenham um tempo de vida maior. Como TPC-C apresenta uma carga de trabalho majoritariamente de leitura (93,72%), isso pode ter contribuído no resultado do seu desempenho, dificultado a classificação de páginas frequentes.

Dos três AM-LRU, EBRES e ARC, o AM-LRU tem uma maior dificuldade, pois não apresenta um mecanismo de histórico, como as páginas *ghost* do EBRES e ARC e o tempo vida das suas páginas em geral também é o menor. Independente do *scan* artificial adicionado, o LIRS apresentou o melhor despenho e AM-LRU o pior se distanciado dos demais, enquanto o

[2] A Figura 54 mostra mais em detalhe – como se fizéssemos um "zoom", como os algoritmos testados se comportam depois do scan.

EBRES ficou com um desempenho moderado em comparação com outros algoritmos diferentes do LIRS e AM-LRU.

## 5.5 Conclusão

Este capítulo abordou uma avaliação experimental comparando a performance dos algoritmos ARC, LRU, EBRES LIRS, AM-LRU e CFDC. Os algoritmos foram examinados usando *traces* sintéticos e reais que exploram diferentes padrões de acesso variando o tamanho do *buffer* em páginas. Na Seção 5.3, para cada *traces* foi comparado o desempenho dos algoritmos no tempo de execução, número de *hits* e número de escritas. Por fim, na Seção 5.4 realizamos testes com o foco na avaliação do mecanismo de *scan resistance* usando o *traces* do TPC-C.

# 6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi proposto o EBRES, uma estratégia de substituição de páginas para gerenciadores de *buffer* de banco de dados. Este capítulo apresenta as conclusões e trabalhos futuros desta dissertação e é dividido da seguinte forma. Na Seção 6.1 é feita uma visão geral desta dissertação. A Seção 6.2 resume os principais resultados. E por fim, a Seção 6.3 é apresentado os trabalhos futuros.

## 6.1 Visão Geral

Para realizar o objetivo geral desta dissertação, que é a elaboração de uma nova estratégia de substituição de páginas de *buffer* foi feito uma revisão com uma breve fundamentação teórica sobre os aspectos e conceitos internos sobre a implementação de gerenciadores de *buffer* de banco de dados. Em seguida, foram apresentados diversos artigos de pesquisa sobre estratégias de substituição de páginas destacando seu fluxo de execução, principais características, vantagens e desvantagens. Essas estratégias evoluíram e se adaptaram a diversos aspectos tecnológicos, como a popularização das memórias flash, o que nos permitiu dividir essas estratégias em dois grupos: políticas de substituição de *buffer* não assimétricas e assimétricas. Diretamente ou indiretamente, muitas dessas estratégias podem estar relacionadas entre si, pois utilizam mecanismos adaptados ou aprimorados para novos contextos como memórias flash. Logo, foi elaborado uma linha do tempo histórica com algumas dessas percepções de relacionamentos entre esses algoritmos. Comparamos estratégias relevantes estudadas nas Seções 3.2 e 3.1 usando diferentes critérios de análise como os seus mecanismos de prevenção de problemas.

A nossa proposta EBRES relaciona as requisições feitas ao *buffer* como uma série temporal individual para cada página. Ele provê uma complexidade constante, para isso, usamos um método simples de predição chamado suavização exponencial aplicado para cada página no *buffer*. EBRES coleta estatísticas como a frequência de leitura e escrita para identificar padrões de acesso e adaptar suas estruturas de dados. Com a expansão das técnicas de predição, podemos estar iniciando uma nova era no projeto dessas estratégias, assim como foi no EBRES, existem vários desafios para desenvolver estratégias com baixo custo computacional que sejam capazes de atender os requisitos críticos de um SGBD.

## 6.2 Resultados principais

A avaliação experimental realizada mostrou resultados sobre o desempenho do EBRES comparado com os algoritmos LIRS, ARC, LRU, CFDC e AM-LRU. Foi analisado a performance dos algoritmos usando *traces* sintéticos e reais que exploram diferentes padrões de acesso. A princípio, para cada algoritmo, variamos o tamanho do *buffer* e examinamos o impacto da carga de trabalho no seu tempo de execução, número de *hits* e número de escritas de cada *traces* experimentados. Em seguida, foi realizado testes mais específicos, avaliando o mecanismo de *scan resistance* de cada algoritmo com o objetivo de observar o impacto no número de *hits* após um longo *scan* de páginas. Neste tipo de experimento, usamos o cenário de pior caso no qual a operação de scan inunda todo o *buffer*.

Durante os primeiros experimentos, EBRES se destaca dos demais algoritmos por manter um equilíbrio nos diferentes cenários testados. Os algoritmos de complexidade não constante AM-LRU, LIRS e CFDC apresentam comportamentos bem mais variantes nos quesitos de *hits*, escritas, e tempo de execução. Já algoritmos de complexidade constante LRU e ARC, embora possam ser competitivos no número de *hits* apresentam dificuldades com o número de escritas. Para os experimentos de *scan resistance* EBRES apresentou uma ótima tolerância a *scans* para *buffer* com tamanhos grandes e médios, apenas em cenários extremos, com tamanhos de *buffer* muito pequenos, apresentou dificuldades.

## 6.3 Trabalhos futuros

Nesta dissertação foi proposto o EBRES, desse modo, alguns desafios e extensões do algoritmo como trabalhos futuros podem ser listados:

- Parâmetros de ajuste dinâmicos: EBRES possui diversos parâmetros de ajuste. Sendo assim, poderíamos está usando valores para estes parâmetros que melhor se adaptem a carga de trabalho atual que o algoritmo está executando. Por exemplo, os tamanhos mínimos das quatro listas principais estão com valor fixo de 10% do tamanho do *buffer* (em páginas). Dependendo da carga de trabalho, seja mais leitura ou escrita ou exigia uma área maior de recência ou frequência, hipoteticamente se esses parâmetros são calculados em tempo de execução baseados no padrão da carga de trabalho, poderíamos melhorar a performance do *buffer*
- Incluir novas estatísticas: EBRES utiliza um mecanismo de coleta de estatísticas no qual

adquire informações temporárias de como está carga de trabalho em execução. Além das estatísticas atuais de porcentagem de requisições de leitura, escrita e miss; outras estatísticas poderiam ser utilizadas. Por exemplo, estatísticas sobre os ID's da páginas próximos são indicativos da ocorrência de um acesso sequencial.

- Investigar outras técnicas de predição: EBRES usa o modelo de suavização exponencial simples. Porém existem outras técnicas de predição ou modelos de suavização exponencial mais complexos e específicos para diferentes tipos de séries temporais, por exemplo séries com tendência ou um componente sazonal. Poderíamos também executar várias técnicas simultaneamente desde que a complexidade destas técnicas não apresente um grande impacto na performance.
- Especializar o algoritmo para uma carga de trabalho analítica: No contexto tecnológico atual, técnicas de análise de dados estão em constante crescimento exigindo cada vez mais um processamento eficiente do banco de dados. Estender o EBRES para atender as demandas deste processamento analítico pode trazer benefícios na área de análise de dados. Por exemplo, o EBRES poderia considerar mais informações disponibilizado pelo processador de consulta (como *hits*) que descrevem qual processamento está ocorrendo nas páginas. Informações se a página é temporária, se a página foi solicitada por um *scan*, se a página tem uma perspectiva de ser referenciada no futuro, etc. Esses exemplos de informações adicionais possivelmente resultariam em decisões mais precisas de substituição.

# REFERÊNCIAS

ARI, I.; AMER, A.; GRAMACY, R. B.; MILLER, E. L.; BRANDT, S. A.; LONG, D. D. Acme: Adaptive caching using multiple experts. In: **WDAS**. [S.l.: s.n.], 2002. v. 2, p. 143–158.

ARLITT, M.; CHERKASOVA, L.; DILLEY, J.; FRIEDRICH, R.; JIN, T. Evaluating content management techniques for web proxy caches. **SIGMETRICS Perform. Eval. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 27, n. 4, p. 3–11, mar. 2000. ISSN 0163-5999. Disponível em: <https://doi.org/10.1145/346000.346003>.

ASSOCIATION, S. N. I. **Microsoft Storage File Server Traces**. 2007. <http://iotta.snia.org/traces/158>. [Online; accessed 04-October-2014].

ASSOCIATION, S. N. I. **TPC-C SNIA Traces**. 2007. <http://iotta.snia.org/traces/131>. [Online; accessed 10-October-2014].

ASSOCIATION, S. N. I. **TPC-E SNIA Traces**. 2007. <http://iotta.snia.org/traces/133>. [Online; accessed 10-October-2014].

BAI, S.; BAI, X.; CHE, X. Window-lrfu: A cache replacement policy subsumes the lru and window-lfu policies. John Wiley and Sons Ltd., GBR, v. 28, n. 9, p. 2670–2684, jun. 2016. ISSN 1532-0626. Disponível em: <https://doi.org/10.1002/cpe.3730>.

BANSAL, S.; MODHA, D. S. Car: Clock with adaptive replacement. In: **Proceedings of the 3rd USENIX Conference on File and Storage Technologies**. USA: USENIX Association, 2004. (FAST '04), p. 187–200.

BARROS, A.; LEHFELD, N. **Fundamentos de Metodologia Cientifica - um Guia Para a Iniciação Cientifica**. [S.l.]: Makron Books, 2000.

Belady, L. A. A study of replacement algorithms for a virtual-storage computer. **IBM Systems Journal**, v. 5, n. 2, p. 78–101, 1966. ISSN 0018-8670.

CHOI, H.; PARK, S. Learning future reference patterns for efficient cache replacement decisions. **IEEE Access**, v. 10, p. 25922–25934, 2022.

CORBATÓ, F.; TECHNOLOGY), P. M. M. I. of. **A Paging Experiment with the Multics System**. [S.l.]: Massachusetts Institute of Technology, 1968. (Project MAC).

DIACONU, C.; FREEDMAN, C.; ISMERT, E.; LARSON, P.; MITTAL, P.; STONECIPHER, R.; VERMA, N.; ZWILLING, M. Hekaton: Sql server's memory-optimized oltp engine. In: **ACM International Conference on Management of Data 2013**. [S.l.: s.n.], 2013.

EFFELSBERG, W.; HäRDER, T. Principles of database buffer management. **ACM Trans. Database Syst.**, ACM, New York, NY, USA, 1984.

ELDAWY, A.; LEVANDOSKI, J.; LARSON, P.-r. Trekking through siberia: Managing cold data in a memory-optimized database. **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 11, p. 931–942, jul 2014. ISSN 2150-8097. Disponível em: <https://doi.org/10.14778/2732967.2732968>.

FAN, Z.; DU, D. H. C.; VOIGT, D. H-arc: A non-volatile memory based cache policy for solid state drives. In: **2014 30th Symposium on Mass Storage Systems and Technologies (MSST)**. [S.l.: s.n.], 2014. p. 1–11.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Implementação de sistemas de banco de dados**. [S.l.]: Campus, 2001.

HARIZOPOULOS, S.; ABADI, D. J.; MADDEN, S.; STONEBRAKER, M. Oltp through the looking glass, and what we found there. In: **Proceedings of the 2008 ACM SIGMOD**. [S.l.: s.n.], 2008. (SIGMOD '08).

HE, J.; JIA, G.; HAN, G.; WANG, H.; YANG, X. Locality-aware replacement algorithm in flash memory to optimize cloud computing for smart factory of industry 4.0. **IEEE Access**, v. 5, p. 16252–16262, 2017.

HELLERSTEIN, J. M.; STONEBRAKER, M.; HAMILTON, J. Architecture of a database system. **Found. Trends databases**, Now Publishers Inc., Hanover, MA, USA, v. 1, n. 2, p. 141–259, fev. 2007. ISSN 1931-7883.

JIANG, S.; CHEN, F.; ZHANG, X. Clock-pro: An effective improvement of the clock replacement. In: **Proceedings of the Annual Conference on USENIX Annual Technical Conference**. USA: USENIX Association, 2005. (ATEC '05), p. 35.

JIANG, S.; ZHANG, X. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: **Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: Association for Computing Machinery, 2002. (SIGMETRICS '02), p. 31–42. ISBN 1581135319. Disponível em: <https://doi.org/10.1145/511334.511340>.

JIN, P.; OU, Y.; HäRDER, T.; LI, Z. Ad-lru: An efficient buffer replacement algorithm for flash-based databases. **Data Knowl. Eng.**, Elsevier Science Publishers B. V., NLD, v. 72, p. 83–102, fev. 2012. ISSN 0169-023X. Disponível em: <https://doi.org/10.1016/j.datak.2011.09.007>.

JO, H.; KANG, J.-U.; PARK, S.-Y.; KIM, J.-S.; LEE, J. Fab: flash-aware buffer management policy for portable media players. **IEEE Transactions on Consumer Electronics**, v. 52, n. 2, p. 485–493, 2006.

JR, E. S. G. Exponential smoothing: The state of the art—part ii. **International journal of forecasting**, Elsevier, v. 22, n. 4, p. 637–666, 2006.

JUNG, H.; SHIM, H.; PARK, S.; KANG, S.; CHA, J. Lru-wsr: integration of lru and writes sequence reordering for flash memory. **IEEE Transactions on Consumer Electronics**, v. 54, n. 3, p. 1215–1223, 2008.

JUNG, H.; YOON, K.; SHIM, H.; PARK, S.; KANG, S.; CHA, J. Lirs-wsr: Integration of lirs and writes sequence reordering for flash memory. In: GERVASI, O.; GAVRILOVA, M. L. (Ed.). **Computational Science and Its Applications – ICCSA 2007**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 224–237. ISBN 978-3-540-74472-6.

KABRA, N.; DEWITT, D. J. Efficient mid-query re-optimization of sub-optimal query execution plans. In: ACM. **ACM SIGMOD Record**. [S.l.], 1998.

KIM, C.-S.; LEE, D.; KIM, J.-H.; CHOI, J.; NOH, S.; MIN, S.; CHO, Y. Lrfu (least recently/frequently used) replacement policy: A spectrum of block replacement policies. In: . [S.l.: s.n.], 1996.

KIM, H.; AHN, S. Bplru: A buffer management scheme for improving random writes in flash storage. In: **Proceedings of the 6th USENIX Conference on File and Storage Technologies**. USA: USENIX Association, 2008. (FAST'08).

LEVANDOSKI, J. J.; LARSON, P.-Å.; STOICA, R. Identifying hot and cold data in main-memory databases. In: IEEE. **2013 IEEE 29th International Conference on Data Engineering (ICDE)**. [S.l.], 2013. p. 26–37.

LI, C. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In: **Proceedings of the 11th ACM International Systems and Storage Conference**. New York, NY, USA: Association for Computing Machinery, 2018. (SYSTOR '18), p. 59–64. ISBN 9781450358491. Disponível em: <https://doi.org/10.1145/3211890.3211891>.

LI, C. Clock-pro+: Improving clock-pro cache replacement with utility-driven adaptation. In: **Proceedings of the 12th ACM International Conference on Systems and Storage**. New York, NY, USA: Association for Computing Machinery, 2019. (SYSTOR '19), p. 1–7. ISBN 9781450367493. Disponível em: <https://doi.org/10.1145/3319647.3325838>.

LI, C.; FENG, D.; HUA, Y.; XIA, W.; WANG, F. Gasa: A new page replacement algorithm for nand flash memory. In: **2016 IEEE International Conference on Networking, Architecture and Storage (NAS)**. [S.l.: s.n.], 2016. p. 1–9.

LI, Z.; JIN, P.; SU, X.; CUI, K.; YUE, L. Ccf-lru: a new buffer replacement algorithm for flash memory. **IEEE Transactions on Consumer Electronics**, v. 55, n. 3, p. 1351–1359, 2009.

LIN, M.; CHEN, S.; WANG, G.; WU, T. Hdc: An adaptive buffer replacement algorithm for nand flash memory-based databases. **Optik**, v. 125, n. 3, p. 1167–1173, 2014. ISSN 0030-4026. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0030402613011959>.

MEGIDDO, N.; MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In: **Proceedings of the 2Nd USENIX Conference on File and Storage Technologies**. [S.l.: s.n.], 2003.

NICOLA, V. F.; DAN, A.; DIAS, D. M. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In: . New York, NY, USA: Association for Computing Machinery, 1992. (SIGMETRICS '92/PERFORMANCE '92), p. 35–46. ISBN 0897915070. Disponível em: <https://doi.org/10.1145/133057.133084>.

ON, S. T.; LI, Y.; HE, B.; WU, M.; LUO, Q.; XU, J. Fd-buffer: A buffer manager for databases on flash disks. In: . New York, NY, USA: Association for Computing Machinery, 2010. (CIKM '10), p. 1297–1300. ISBN 9781450300995. Disponível em: <https://doi.org/10.1145/1871437.1871605>.

O'NEIL, E. J.; O'NEIL, P. E.; WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. **SIGMOD Rec.**, ACM, New York, NY, USA, v. 22, n. 2, p. 297–306, jun. 1993. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/170036.170081>.

ORACLE. **MySQL 8.0 Reference Manual**. 2021. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html>.

ORACLE. **Oracle - Database Concepts**. 2021. Disponível em: <https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/memory-architecture.html#GUID-AA95A3C7-C452-4DDB-B7E4-44CB43B1E9C3>.

OU, Y.; HäRDER, T. Clean first or dirty first? a cost-aware self-adaptive buffer replacement policy. In: . New York, NY, USA: Association for Computing Machinery, 2010. (IDEAS '10), p. 7–14. ISBN 9781605589008. Disponível em: <https://doi.org/10.1145/1866480.1866482>.

OU, Y.; HäRDER, T.; JIN, P. Cfdc: A flash-aware replacement policy for database buffer management. In: **Proceedings of the Fifth International Workshop on Data Management on New Hardware**. New York, NY, USA: Association for Computing Machinery, 2009. (DaMoN '09), p. 15–20. ISBN 9781605587011. Disponível em: <https://doi.org/10.1145/1565694.1565698>.

OU, Y.; JIN, P.; HäRDER, T. Flash-aware buffer management for database systems. **Int. J. Knowledge-Based Organ.**, IGI Global, USA, v. 3, n. 4, p. 22–39, out. 2013. ISSN 2155-6393. Disponível em: <https://doi.org/10.4018/ijkbo.2013100102>.

PARK, S.; KIM, Y.; URGAONKAR, B.; LEE, J.; SEO, E. A comprehensive study of energy efficiency and performance of flash-based ssd. In: . [S.l.: s.n.], 2011. v. 57, n. 4, p. 354–365. ISSN 1383-7621.

PARK, S.-y.; JUNG, D.; KANG, J.-u.; KIM, J.-s.; LEE, J. Cflru: A replacement algorithm for flash memory. In: **Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems**. New York, NY, USA: Association for Computing Machinery, 2006. p. 234–241. ISBN 1595935436. Disponível em: <https://doi.org/10.1145/1176760.1176789>.

ROBINSON, J. T.; DEVARAKONDA, M. V. Data cache management using frequency-based replacement. In: **Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: Association for Computing Machinery, 1990. (SIGMETRICS '90), p. 134–142. ISBN 0897913590. Disponível em: <https://doi.org/10.1145/98457.98523>.

RODRIGUEZ, L. V.; YUSUF, F.; LYONS, S.; PAZ, E.; RANGASWAMI, R.; LIU, J.; ZHAO, M.; NARASIMHAN, G. Learning cache replacement with CACHEUS. In: **19th USENIX Conference on File and Storage Technologies (FAST 21)**. USENIX Association, 2021. p. 341–354. ISBN 978-1-939133-20-5. Disponível em: <https://www.usenix.org/conference/fast21/presentation/rodriguez>.

SHASHA, D.; JOHNSON, T. 2q: A low overhead high performance buffer management replacement algoritm. In: **Proceedings of the Twentieth International Conference on Very Large Databases, Santiago, Chile**. [S.l.: s.n.], 1994. p. 439–450.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts, 5th Edition**. [S.l.]: McGraw-Hill Book Company, 2005.

SORJAMAA, A.; HAO, J.; REYHANI, N.; JI, Y.; LENDASSE, A. Methodology for long-term prediction of time series. **Neurocomputing**, Elsevier, v. 70, n. 16-18, p. 2861–2869, 2007.

STOICA, R.; AILAMAKI, A. Enabling efficient os paging for main-memory oltp databases. In: **Proceedings of the Ninth International Workshop on Data Management on New Hardware**. New York, NY, USA: Association for Computing Machinery, 2013. (DaMoN '13). ISBN 9781450321969. Disponível em: <https://doi.org/10.1145/2485278.2485285>.

TAVARES, J. A. **Database Buffer Management Strategies for Asymmetric Media**. Tese (Master dissertation) — Universidade de Fortaleza - Unifor, 2015.

TPC. **BenchmarkTM C.** 2010. <http://www.tpc.org/tpcc/>. Acessado em: 22/08/2022.

VERÍSSIMO, A. J.; ALVES, C. da C.; HENNING, E.; AMARAL, C. E. do; CRUZ, A. C. da. Métodos estatísticos de suavização exponencial holt-winters para previsão de demanda em uma empresa do setor metal mecânico. **Revista Gestão Industrial**, v. 8, n. 4, 2013.

VIETRI, G.; RODRIGUEZ, L. V.; MARTINEZ, W. A.; LYONS, S.; LIU, J.; RANGASWAMI, R.; ZHAO, M.; NARASIMHAN, G. Driving cache replacement with ml-based lecar. In: **Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems**. USA: USENIX Association, 2018. (HotStorage'18), p. 3.

VOLTACTIVEDATA. **VOLT ACTIVE DATA**. 2010. <https://www.voltactivedata.com/>. Acessado em: 12/09/2022.

WANG, F.; JIANG, X.; HUANG, J.; CHEN, F. Pa-lirs: An adaptive page replacement algorithm for nand flash memory. **Electronics**, v. 9, n. 12, 2020. ISSN 2079-9292. Disponível em: <https://www.mdpi.com/2079-9292/9/12/2172>.

WU, X.; CAI, D.; GUAN, S. A multiple LRU list buffer management algorithm. **IOP Conference Series: Materials Science and Engineering**, IOP Publishing, v. 569, p. 052002, aug 2019. Disponível em: <https://doi.org/10.1088/1757-899x/569/5/052002>.

XIA, Z.; BU, T. The implementation of flash-aware buffer replacement algorithms in postgresql. In: **2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)**. [S.l.: s.n.], 2015. p. 1215–1219.

ZHANG, W. E.; SHENG, Q. Z.; TAYLOR, K.; QIN, Y. Identifying and caching hot triples for efficient rdf query processing. In: RENZ, M.; SHAHABI, C.; ZHOU, X.; CHEEMA, M. A. (Ed.). **Database Systems for Advanced Applications**. Cham: Springer International Publishing, 2015. p. 259–274. ISBN 978-3-319-18123-3.

ZHOU, Y.; PHILBIN, J.; LI, K. The multi-queue replacement algorithm for second level buffer caches. In: **Proceedings of the General Track: 2001 USENIX Annual Technical Conference**. USA: USENIX Association, 2001. p. 91–104. ISBN 188044609X.