Algoritmos para Identificação de Dados Frios em Bancos de Dados em Memória

Alessandra V. Santos[1], **Vládia Pinheiro**[1], **José Maria Monteiro**[2]

[1]Centro de Tecnologia
Universidade de Fortaleza (UNIFOR)

[2]Departamento de Computação
Universidade Federal do Ceará (UFC)

alessa_santos@yahoo.com.br, vladiacelia@unifor.br, monteiro@dc.ufc.br

Abstract. *The growth in main-memory storage capacity has fueled the development of main-memory database systems. Thus, many OLTP databases can be stored entirely in the main memory. However, due to the continued growth of data, dealing with data overflow is crucial. OLTP workloads often exhibit skewed access patterns, where some records are hot (frequently accessed) but many records are cold (rarely or never accessed). So, it is more economical to store the coldest records on secondary storage such as flash or hard disk. Recently, many research works have addressed the data overflow problem, developing approaches to identify hot/cold data. In this paper, we present two new algorithms called 2QCold and ARCold, which adapt the classic 2Q and ARC cache algorithms to identify cold data. We implement our algorithms using Seal-DB and compare them with the classic LRU, Forward and Belady algorithms. The TPC-C benchmark was used in the experiments. The results show that both 2QCold and ARCold reduce response time and increase hit ratio outperforming related works.*

Resumo. *O crescimento da capacidade de armazenamento da memória principal impulsionou o desenvolvimento de sistemas de banco de dados da memória principal. Assim, muitos bancos de dados OLTP podem ser armazenados inteiramente na memória principal. No entanto, devido ao crescimento contínuo de dados, é crucial lidar com o transbordamento de dados. As cargas de trabalho OLTP geralmente exibem padrões de acesso onde alguns registros são quentes (acessados com frequência), mas muitos registros são frios (raramente ou nunca acessados). Portanto, é mais econômico armazenar os registros mais frios no armazenamento secundário, como flash ou disco rígido. Recentemente, muitos trabalhos de pesquisa abordaram o problema de transbordamento de dados, desenvolvendo abordagens para identificar dados quentes/frios. Neste artigo, apresentamos dois novos algoritmos chamados 2QCold e ARCold, que adaptam os algoritmos clássicos de cache 2Q e ARC para identificar dados frios. Implementamos nossos algoritmos usando Seal-DB e os comparamos com os algoritmos clássicos LRU, Forward e Belady. O benchmark TPC-C foi usado nos experimentos. Os resultados mostram que tanto 2QCold quanto ARCold reduzem o tempo de resposta e aumentam a taxa de acerto superando os trabalhos relacionados.*

1. Introdução

O crescimento da capacidade de armazenamento da memória principal impulsionou o desenvolvimento de sistemas de banco de dados da memória principal (*In-Memory Databases - IMDB*). Os IMDBs surgiram com a finalidade de atender um conjunto de aplicações que possuem como requisito processar grandes volumes de dados em tempo real. Atualmente, muitos bancos de dados OLTP (*Online Transaction Processing*) podem ser armazenados inteiramente na memória principal. No entanto, algumas aplicações possuem como característica o crescimento contínuo dos dados a serem gerenciados. É o caso das aplicações de *Big Data* [Emmanuel and Stanier 2016]. Desta forma, torna-se crucial lidar com o fenômeno do transbordamento de dados (*data overflow*), o qual ocorre quando o tamanho dos dados excede o tamanho da memória principal. As cargas de trabalho OLTP geralmente exibem padrões de acesso onde alguns registros são "quentes" (acessados com frequência), mas muitos registros são "frios" (raramente ou nunca acessados). Portanto, é mais econômico armazenar os registros mais frios no armazenamento secundário.

Recentemente, algumas pesquisas abordaram o problema do transbordamento de dados, desenvolvendo abordagens para identificar dados quentes/frios. O objetivo principal destes trabalhos é separar os dados "quentes" (frequentes) e "frios" (não frequentes). A principal diferença entre as abordagens existentes é o nível de granularidade em que os dados são acessados e classificados como quentes ou frios, a saber: nível da tupla, nível de página ou nível de atributo. A maioria dessas abordagens usa as técnicas LRU (*Least Recently Used*) e LFU (*Least Frequently Used*) para distinguir entre dados quentes e frios. Com essa informação é possível tomar uma decisão a fim de evitar o transbordamento de dados na memória. Os dados frios podem ser despejados em disco, compactados ou mesmo eliminados, liberando espaço na memória principal. Porém, essa decisão, bem como sua implementação, está fora do escopo deste trabalho.

Neste artigo, apresentamos dois novos algoritmos denominados *2QCold* e *ARCold*, que adaptam os algoritmos clássicos de cache *2Q* e *ARC* (*Adaptive Replacement Cache*) para identificar dados frios. Implementamos os algoritmos propostos utilizando o *Seal-DB* [Moraes et al. 2017] e os comparamos com os algoritmos clássicos *LRU*, *Forward* [Stoica et al. 2013] e *Belady* [Belady 1966]. O *benchmark* TPC-C foi usado nos experimentos. Os resultados mostram que tanto *2QCold* quanto *ARCold* reduzem o tempo de resposta e aumentam a taxa de acerto superando os trabalhos relacionados. Adicionalmente, os algoritmos *2QCold* e *ARCold* podem ser configurados para executar tanto de forma *online* quanto *offline*. No método *online*, os algoritmos propostos coletam e utilizam informações acerca das consultas que estão sendo executadas pelo IMDB. Já no método *offline*, os algoritmos utilizam as informações armazenadas nos arquivos de *log*, acerca das consultas executadas anteriormente. Os experimentos realizados mostraram que os algoritmos *2QCold* e *ARCold* em suas versões *online* e *offline* reduziram o tempo de resposta do sistema em 5% e 28%, respectivamente. Além disso, ambos conseguiram aumentar a taxa de acerto na identificação dos dados frios em 27% em relação aos trabalhos relacionados.

O restante desse artigo está organizado da seguinte forma, A Seção 2 apresenta os trabalhos relacionados. Na Seção 3 são apresentados os algoritmos propostos para identificação de dados frios. Na Seção 4 os experimentos realizados e os resultados obtidos são discutidos. A Seção 5 conclui este artigo e aponta possíveis trabalhos futuros.

2. Trabalhos Relacionados

O gerenciador de bancos de dados em memória TimesTen [Lahiri et al. 2013] implementa duas políticas de envelhecimento dos dados. Tais políticas definem como os dados armazenados na memória RAM serão movidos para disco em caso de falta de espaço na memória principal. A primeira política utiliza o algoritmo *LRU* (*Least Recently Used*), removendo os itens que estão há mais tempo sem uso (acesso), dentro de um intervalo de tempo especificado pelo SGBD. A segunda política utiliza uma estrutura *FIFO*, removendo os dados mais antigos, ou seja, que foram levados para a memória a mais tempo.

Kemper et al. [Kemper Alfons 2011] propuseram um banco de dados em memória híbrido, chamado HyPer. No HyPer, os dados são armazenados em duas partições, de acordo com a sua frequência de acesso, denominadas partição fria e partição quente. Quando uma determinada tupla t, armazenada na partição fria, é acessada, esta é movida para a partição quente. Eventualmente, uma tupla s armazenada na partição quente é selecionada e movida para a partição fria. Assim, ocorre uma troca de partição entre as tuplas t e s. A partição fria é armazenada em páginas "grandes" de 2 MB e a partição quente é armazenada em páginas "pequenas" de 4 KB. Para otimizar o armazenamento do Hyper, Funke et al. [Funke Florian 2012] propuseram um mecanismo de compactação dos dados frios utilizando um componente de monitoramento chamado *Access Observer*. Este mecanismo determina quais tuplas devem ser consideradas quentes e quais serão consideradas frias. O armazenamento das tuplas quentes utiliza uma estrutura clusterizada. Já os dados frios são compactos e posteriormente armazenados em disco.

No Hekaton [Diaconu et al. 2013], as tabelas e os índices ficam armazenados na memória principal. O projeto Siberia [Eldawy et al. 2014] surgiu com o objetivo de otimizar o armazenamento de dados quentes e frios no Hekaton. Para identificar e classificar dados quentes e frios, o Projeto Siberia utilizou uma abordagem não intrusiva e *offline*, baseada na análise de *logs* e na estimativa de frequências por meio de uma técnica chamada *exponential smoothing*.

Em [Pathak et al. 2018], os autores apresentaram vários esquemas para ILM (*Information Life Cycle Management*) com a finalidade de reter apenas os dados quentes na memória principal e armazenar os dados mais frios na memória secundária. As técnicas propostas baseiam-se nas características da carga de trabalho e no particionamento de tabelas. Assim, as partições que armazenam dados quentes são mantidas na memória principal enquanto as partições que armazenam dados frios são mantidas em memória secundária. As técnicas propostas foram implementadas no sistema de gerenciamento de bancos de dados SAP ASE.

O algoritmo *HC Apriori*, uma adaptação do Apriori, foi proposto por Afify et al. [Afify Ghada M 2016] com a finalidade de classificar as tuplas em quentes (acessadas com frequência) ou frias (acessadas com pouca frequência). O *HC Apriori* utiliza técnicas de mineração de conjunto de itens frequentes (*Frequent Item set Mining - FIM*). Em [Ha et al. 2021], os autores propuseram um esquema dinâmico de identificação de dados quentes e frios, o qual se adapta às cargas de trabalho. Assim, dependendo do tipo da carga de trabalho, o mecanismo atribui-se um peso maior ou menor à recência e à frequência. O esquema proposto adota vários *bloom filters* para representar a recência e à frequência.

3. Algoritmos Propostos: 2QCold e ARCold

Nesta seção, iremos apresentar dois algoritmos (denominados *2QCold* e *ARCold*) capazes de identificar dados frios em bancos de dados em memória (IMDBs). Os algoritmos *2QCold* e *ARCold* podem ser configurados para executar tanto de forma *online* quanto *offline*. No método *online*, os algoritmos coletam informações acerca das consultas que estão sendo executadas. Já no método *offline*, os algoritmos utilizam as informações armazenadas nos arquivos de *log*, acerca das consultas executadas anteriormente.

3.1. Algoritmo 2QCold

O *2QCold* é uma adaptação do algoritmo *2Q*, proposto por Jonhson et al [Johnson Theodore 1994] no contexto do gerenciamento de *cache*. O *2Q* foi escolhido como base para o *2QCold* por apresentar melhor desempenho que os algoritmos LRU e LRU-K, tendo uma baixa sobrecarga e rápida execução. O algoritmo *2QCold* utiliza três filas (*A1in*, *A1out* e *Cold*) e uma lista (*Am*) para gerenciar os dados armazenados no IMDB. Estas estruturas armazenam ponteiros de tuplas, os quais identificam onde estas tuplas estão armazenadas na memória RAM. A seguir, iremos descrever em detalhes o funcionamento dessas estruturas.

A1in é a fila de entrada do algoritmo *2QCold*. Sua finalidade é armazenar os ponteiros para as tuplas acessadas pela primeira vez ou que são requisitadas com frequência. Assim, quando uma tupla é inicialmente requisitada, um ponteiro para esta tupla é inserido na cabeça da fila *A1in*. A fila *A1in* é uma estrutura do tipo FIFO (*First In First Out*). Assim, quando a fila *A1in* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o seu último ponteiro, ou seja, o ponteiro que está na cauda da fila (elemento mais antigo) é movido para a fila *A1out*. Sempre que uma tupla cujo ponteiro está armazenado em *A1in* é acessada, seu ponteiro permanece exatamente na mesma posição. Como esta é uma fila de entrada, o que determina a transferência de um ponteiro de tupla para a fila *A1out* é a sua recência e não sua frequência de acesso. A literatura recomenda que o seu tamanho seja definido entre 20-30% do tamanho do banco de dados [Johnson Theodore 1994].

A fila *A1out* recebe ponteiros para tuplas que estavam armazenados em *A1in*, mas que por falta de espaço precisaram ser movidos, utilizando-se o critério de recência. Sua estrutura também é do tipo FIFO. Porém, como uma fila de saída, ou seja de armazenamento temporário, a ideia é que os ponteiros de tupla permaneçam nesta fila por pouco tempo. Assim, sempre que uma tupla cujo ponteiro está armazenado em *A1out* é acessada, seu ponteiro é movido para a lista *Am*. Além disso, quando a fila *A1out* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o seu último ponteiro (elemento mais antigo) é movido para a fila *Cold*.

A lista *Am* tem por finalidade armazenar os ponteiros para as tuplas que são acessadas com frequência e por um longo período de tempo, ou seja, tuplas quentes. A lista *Am* utiliza o algoritmo LRU. Assim, sempre que uma tupla cujo ponteiro está armazenado em *Am* é acessada, seu ponteiro é movido para o início da lista. Assim, tuplas que são frequentemente acessadas tendem a permanecer nesta lista. Adicionalmente, quando a lista *Am* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o seu último ponteiro é movido para a fila *Cold*. O tamanho da lista *Am* é definido inicialmente em 40% do tamanho do banco de dados.

A fila *Cold* se comporta como uma estrutura do tipo FIFO e é responsável por armazenar os ponteiros de tuplas que não estão sendo mais acessadas, ou seja, tuplas frias, oriundas da fila *A1out* e da lista *Am*. O tamanho da fila *Cold* é definido inicialmente em 20% do tamanho do banco de dados, representados em quantidade de ponteiro de tuplas.

Os tamanhos das filas *A1in*, *A1out* e *Cold*, bem como da lista *Am* são ajustados de acordo com a carga de dados. O parâmetro K determina a quantidade de ponteiros utilizada para identificar tuplas quentes. Assim, o tamanho da fila *A1in* mais o tamanho da lista *Am* é igual a K. Desta forma, o tamanho da lista *Am* e da fila *A1in* podem ser dinamicamente ajustados, de acordo com o comportamento de acesso dos dados. A soma dos tamanhos das estruturas *A1in*, *A1out*, *Am* e *Cold* deve ser igual à quantidade de ponteiros de tuplas armazenadas no banco de dados. A variação do tamanho mínimo e máximo das estruturas *A1in*, *A1out* e *Am* se mantiveram semelhantes ao utilizado no algoritmo *2Q* [Johnson Theodore 1994]. A fila *Cold* teve seu tamanho máximo definido de acordo com o total de ponteiros de tuplas armazenadas no banco que dados menos a soma do tamanho das estruturas *A1in*, *A1out* e *Am*. Todas as listas, contendo os ponteiros de tuplas, permanecem em memória. Assim, o SGBD pode tomar decisões de forma extremamente rápida, caso necessite despejar, comprimir ou mesmo eliminar dados frios, a fim de liberar espaço de armazenamento, evitando o transbordamento dos dados.

O recalculo dos tamanhos das estruturas *A1in*, *A1out*, *Am* e *Cold* pode acontecer em dois momentos, os quais serão detalhados a seguir. Quando a fila *Cold* precisa de espaço e a lista *Am* não tem mais espaço para ceder. Nessa situação as estruturas estão cheias. Isso significa que o banco de dados está recebendo a inserção de novas tuplas e há área de armazenamento livre no banco de dados. Ocorre também quando atinge-se o limite determinado ao banco de dados em relação a sua ocupação e para isso é necessário que seja tomada uma ação em relação aos dados identificados como frios. Em ambos os momentos é realizado o recalculo do tamanho das estruturas.

O Algoritmo 1 descreve o funcionamento do *2QCold*. Vale destacar que o algoritmo *2QCold* tem como complexidade O(n), de forma similar ao *2Q* [Johnson Theodore 1994]. O algoritmo *2QCold* (Algoritmo 1) tem como parâmetros de entrada: *kin* (tamanho máximo da fila *A1in*, *kout* (tamanho máximo da fila *A1out*) e *K* (tamanho da fila *A1in* mais o tamanho da lista *Am*). Quando uma tupla é acessada (requisitada), o procedimento *REQUEST* é executado, recebendo como parâmetro de entrada o (*id*) da tupla requisitada (representada por *x*). Inicialmente, se o ponteiro de tupla *x* está na fila *Cold* ele deverá ser movido para a fila *A1in*. Porém, a fila *A1in* pode estar completamente preenchida, ou seja, sem espaço para armazenar *x*. Para tratar este possível cenário, o procedimento *REPLACEMENT* é executado.

O procedimento *REPLACEMENT* avalia, inicialmente, se há espaço livre na área reservada para as estruturas *A1in* e *Am*, em conjunto (linha 19). Caso, ainda haja espaço nada é executado e o procedimento é finalizado. Caso contrário (tamanho de *A1in* + *Am* = K), o procedimento verifica se o tamanho da fila *A1in* chegou ao seu limite (*kin*) (linha 20). Caso, ainda haja espaço na fila *A1in* isto implica que a lista *Am* está ocupando espaço da fila *A1in*. Desta forma, move-se o último elemento da lista *Am* (elemento menos recentemente utilizado) para a fila *A1out* (linhas 27 e 28), a fim de liberar espaço na área reservada para as estruturas *A1in* e *Am*. Caso, não haja espaço na fila *A1in*, será necessário

mover um ponteiro de *A1in* para *A1out*. Porém, é necessário verificar se existe espaço na fila *A1out* (linha 21). Caso ainda haja espaço nesta fila, move-se o primeiro elemento da fila *A1in* (mais antigo na fila) para *A1out* (linhas 24 e 25). Caso contrário, move-se o primeiro ponteiro da fila *A1out* para a fila *Cold*, a fim de liberar espaço em *A1out* (linhas 22 e 23).

Se o ponteiro de tupla *x* está na fila *A1in* nenhuma movimentação precisa ser executada (linha 9). Já se o ponteiro de tupla *x* está na lista *Am*, ele é removido e posteriormente inserido em *Am* com a finalidade de mover o ponteiro *x* para a primeira posição da lista *Am*, o que irá manter este ponteiro mais tempo em *Am*, uma vez que esta lista utiliza o algoritmo *LRU*. Se o ponteiro de tupla *x* está na fila *A1out*, ele deverá ser movido para a lista *Am*. Porém, a lista *Am* pode estar completamente preenchida, ou seja, sem espaço para armazenar *x*. Para tratar este possível cenário, o procedimento *REPLACEMENT* é executado.

Algorithm 1 2QCold

```
1:  kin - tamanho máximo de A1in
2:  kout - tamanho máximo de A1out
3:  K - quantidade de tuplas quentes para identificar
4:  procedure REQUEST(id)
5:      x ← FIND(id)
6:      if x ∈ cold then
7:          REPLACEMENT()
8:          A1in.ADD(x)
9:      if x ∈ A1in then
10:         //do nothing
11:     if x ∈ Am then
12:         Am.REMOVE(x)
13:         Am.ADD(x)
14:     if x ∈ A1out then
15:         A1out.REMOVE(x)
16:         REPLACEMENT()
17:         Am.ADD(x)
18: procedure REPLACEMENT
19:     if A1in.SIZE() + Am.SIZE() = K then
20:         if A1in.SIZE() ≥ kin then
21:             if A1out.SIZE() = kout then
22:                 victim ← A1out.REMOVE_FIRST()
23:                 cold.ADD(victim)
24:             outer ← A1in.REMOVE_FIRST()
25:             A1out.ADD(outer)
26:         else
27:             victim ← Am.REMOVE_FIRST()
28:             A1out.ADD(victim)
```

3.2. Algorimo ARCold

O *ARCold* é uma adaptação do algoritmo *ARC* (*Adaptive Replacement Cache*), proposto por Megiddo et al. [Megiddo and Modha 2003] em 2003 no contexto do gerenciamento de *cache*. Decidimos adaptar o *ARC* por este ser um algoritmo adaptativo, com alta taxa de acerto e que consegue equilibrar os padrões de acesso dos dados de acordo com a recência e a frequência. O algoritmo *ARCold* utiliza três filas (*B1*, *B2* e *Cold*) e duas listas (*T1* e *T2*). Estas estruturas armazenam ponteiros de tuplas, os quais identificam onde estas tuplas estão armazenadas na memória RAM. A seguir, iremos descrever em detalhes o funcionamento dessas estruturas.

A lista *T1* e fila *B1* são responsáveis por armazenar os dados recentemente acessados, sendo que *B1* recebe os dados que não poder ser armazenados em *T1* devido a falta de espaço. A lista *T2* e fila *B2* são responsáveis por armazenar os dados mais frequentes acessados, sendo que *B2* recebe os dados que não poder ser armazenados em *T2* devido a falta de espaço. Na medida em que novos dados vão sendo acessados e, consequentemente, levados para a memória RAM, as filas *B1* e/ou *B2* podem ficar sem espaço livre. Neste caso, os dados nelas armazenados são candidatos a serem movidos para a fila *Cold*. A adaptação adicionada pelo algoritmo *ARCold* não altera a complexidade do algoritmo *ARC*, que é *O(n)* [Megiddo and Modha 2003]. A seguir, iremos discutir em detalhes o funcionamento das estruturas *T1*, *T2*, *B1*, *B2* e *Cold*.

A lista *T1* é a estrutura de entrada do algoritmo *ARCold*. Sua finalidade é armazenar os ponteiros para as tuplas acessadas pela primeira vez. Assim, quando uma tupla é inicialmente requisitada, um ponteiro para esta tupla é armazenado na lista *T1*. Quando a lista *T1* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o algoritmo *LRU* (Least Recently Used) é utilizado para selecionar o ponteiro da tupla mais antiga em *T1*, o qual será movido para a fila *B1*. Sempre que uma tupla cujo ponteiro está armazenado em *T1* é acessada, seu ponteiro é movido para o inicio da lista *T2*. A literatura recomenda que o seu tamanho seja definido em 20% do tamanho do banco de dados [Megiddo and Modha 2003], conforme a quantidade de ponteiro de tuplas.

A lista *T2* tem por finalidade armazenar os ponteiros para as tuplas que são acessadas com frequência e por um longo período de tempo, ou seja, tuplas quentes. Assim, sempre que uma tupla cujo ponteiro está armazenado em *T2* é acessada, seu ponteiro é movido para o início da lista. Assim, tuplas que são frequentemente acessadas tendem a permanecer nesta lista. Adicionalmente, quando a lista *T2* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o algoritmo *LRU* (Least Recently Used) é utilizado para selecionar o ponteiro da tupla mais antiga em *T2*, o qual será movido para a fila *B2*. O tamanho da lista *T2* é definido inicialmente em 30% do tamanho do banco de dados, representados em quantidade de ponteiro de tuplas.

A fila *B1* recebe ponteiros para tuplas que estavam armazenados em *T1*, mas que por falta de espaço precisaram ser movidos, utilizando-se o critério de recência. Quando uma tupla cujo ponteiro está armazenado em *B1* é acessada, seu ponteiro é movido para a lista *T2*. Além disso, quando a fila *B1* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o seu último ponteiro é movido para a fila *Cold*. Como *B1* é uma estrutura do tipo *FIFO*, o último ponteiro corresponde à sua entrada mais antiga.

A fila *B2* recebe ponteiros para tuplas que estavam armazenados em *T2*, mas que por falta de espaço precisaram ser movidos, utilizando-se o critério de recência. Quando uma tupla cujo ponteiro está armazenado em *B2* é acessada, seu ponteiro é movido para a lista *T2*. Além disso, quando a fila *B2* está completamente preenchida e um novo ponteiro de tupla precisa ser nela adicionada, o seu último ponteiro é movido para a fila *Cold*. Como *B2* é uma estrutura do tipo *FIFO*, o último ponteiro corresponde à sua entrada mais antiga. Os tamanhos das filas *B1* e *B2* são definidos inicialmente em 15% do tamanho do banco de dados, em quantidade de ponteiro de tuplas, conforme sugerido em [Megiddo and Modha 2003].

A fila *Cold* se comporta como uma estrutura do tipo FIFO e é responsável por armazenar os ponteiros de tuplas que não estão sendo mais acessadas, ou seja, tuplas frias, oriundas das filas *B1* e *B2*. O tamanho da fila *Cold* é definido inicialmente em 20% do tamanho do banco de dados, representados em quantidade de ponteiro de tuplas. Quando uma tupla cujo ponteiro está armazenado na fila *Cold* é acessada, seu ponteiro é movido para a lista *T2*. Caso um ponteiro de tupla seja movido para a fila *Cold* e a mesma não tenha espaço livre, realiza-se o ajuste do tamanho da fila utilizando parte do espaço reservado para as listas *T1* ou *T2*. Caso *T1* e *T2* estejam completamente preenchidas, faz-se necessário recalcular o tamanho de todas as estruturas utilizadas pelo algoritmo *ARCold*.

O Algoritmo 2 descreve o funcionamento do *ARCold*. Vale destacar que o algoritmo *ARCold* tem como complexidade O(n), de forma similar ao *ARC* [Megiddo and Modha 2003]. Já o algoritmo 3 ilustra os procedimentos de inserção de um ponteiro de tupla nas estruturas *T1*, *T2*, *B1* e *B2*, respectivamente.

Algorithm 2 ARCold

```
1:  procedure REQUEST(id)
2:      x ← FIND(id)
3:      if x = Null then
4:          insertT1(x)
5:      if x ∈ cold then
6:          cold.REMOVE(x)
7:          insertT1(x)
8:      if x ∈ T1 then
9:          T1.REMOVE(x)
10:         insertT2(x)
11:     if x ∈ T2 then
12:         T2.REMOVE(x)
13:         insertT2(x)
14:     if x ∈ B1 then
15:         B1.REMOVE(x)
16:         insertT2(x)
17:     if x ∈ B2 then
18:         B2.REMOVE(x)
19:         insertT2(x)
```

Algorithm 3 Insert Procedures

```
1:  procedure INSERTT1(x)
2:      if T1.full() then
3:          victim ← T1.REMOVE_FIRST()
4:          insertB1.ADD(victim)
5:      T1.ADD(x)
6:  procedure INSERTT2(x)
7:      if T2.full() then
8:          victim ← T2.REMOVE_FIRST()
9:          insertB2.ADD(victim)
10:     T2.ADD(x)
11: procedure INSERTB1(x)
12:     if B1.full() then
13:         victim ← B1.REMOVE_FIRST()
14:         cold.ADD(victim)
15:     B1.ADD(x)
16: procedure INSERTB2(x)
17:     if B2.full() then
18:         victim ← B2.REMOVE_FIRST()
19:         cold.ADD(victim)
20:     B2.ADD(x)
```

4. Avaliação Experimental

Esta seção descreve os experimentos realizados e os resultados obtidos com a finalidade de avaliar o desempenho dos algoritmos propostos para identificação de dados frios: *2QCold* e *ARCold*. O desempenho foi avaliado em duas dimensões: tempo de resposta e taxa de acerto. Além disso, para analisar cada dimensão utilizamos dois cenários de execução: *online* e *offline*. No cenário *online*, os algoritmos de identificação de dados frios coletam e utilizam informações acerca das consultas que estão sendo executadas pelo IMDB. Já no cenário *offline*, os algoritmos utilizam as informações armazenadas nos arquivos de *log*, acerca das consultas executadas anteriormente. Implementamos nossos algoritmos usando *Seal-DB* [Moraes et al. 2017] e os comparamos com os algoritmos clássicos *LRU*, *Forward* [Stoica et al. 2013] e *Belady* [Belady 1966]. O *benchmark* TPC-C [1] fator 1 foi usado nos experimentos por ser um benchmark que trabalha com transações, comportando todos os dados acessados na memória principal.

4.1. Ambiente de Experimentação

Todos os experimentos foram executados em um computador com um processador Intel Core i5, com 8 GB de memória RAM e SSD de 500GB. A carga de trabalho utilizada foi o *Benchmark* TPC-C fator 1. O banco de dados utilizado nos experimentos foi o *Seal-DB* [Moraes et al. 2017], que tem como propósito apoiar o ensino de banco de dados. Adaptamos para funcionar todo em memória principal, incluindo os algoritmos propostos

[1] Benchmark TPC-C: http://www.tpc.org/tpcc/

para realizar a identificação dos dados processados. Nos experimentos foram realizadas três rodadas de execução. Em cada rodada, cada um dos algoritmos foi executado oito vezes. Em seguida, as médias do tempo de resposta e taxa de acerto foram calculadas. Variamos o parâmetro K utilizando os seguintes valores: 1K, 10K, 50K, 100K, 200K, 400K e 600K. Para avaliar o cenário *online* utilizamos os algoritmos *2QCold*, *ARCold* e *LRU*. Já para analisar o cenário *offline* utilizamos os algoritmos *2QCold*, *ARCold* e *Forward* [Stoica et al. 2013]. Adicionalmente, para avaliar a taxa de acerto utilizamos ainda o algoritmo ótimo de Belady [Belady 1966].

4.2. Resultados

4.2.1. Tempo de Resposta

A Figura 1 ilustra os tempos de resposta (em milissegundos) obtidos pelos algoritmos *2QCold*, *ARCold* e *LRU* no cenário *online*. Analisando a Figura 1 podemos observar que os algoritmos *2QCold* e *ARCold* obtiveram uma redução no tempo de resposta de cerca de 1% e 5%, respectivamente, em comparação com o algoritmo de *LRU*.

A Figura 2 ilustra os tempos de resposta (em milissegundos) obtidos pelos algoritmos *2QCold*, *ARCold* e *Forward* [Stoica et al. 2013] no cenário *offline*. Analisando a Figura 2 podemos observar que o algoritmo *2QCold* apresentou uma redução no tempo de resposta que varia entre 17% e 28%, em comparação com o algoritmo de *Forward* [Stoica et al. 2013]. Já o algoritmo *ARCold* obteve uma redução no tempo de resposta que varia entre 17% e 23%.

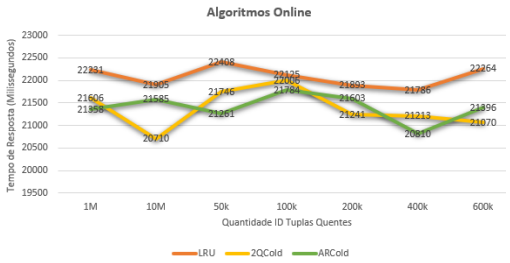


Figura 1. Tempo de Resposta em Milissegundos (Cenário Online)

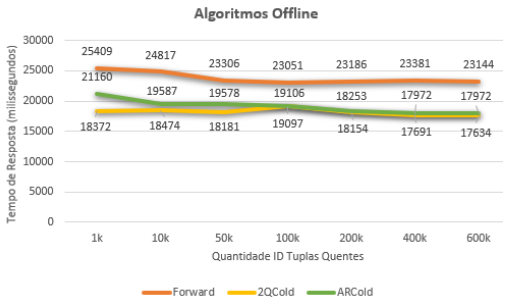
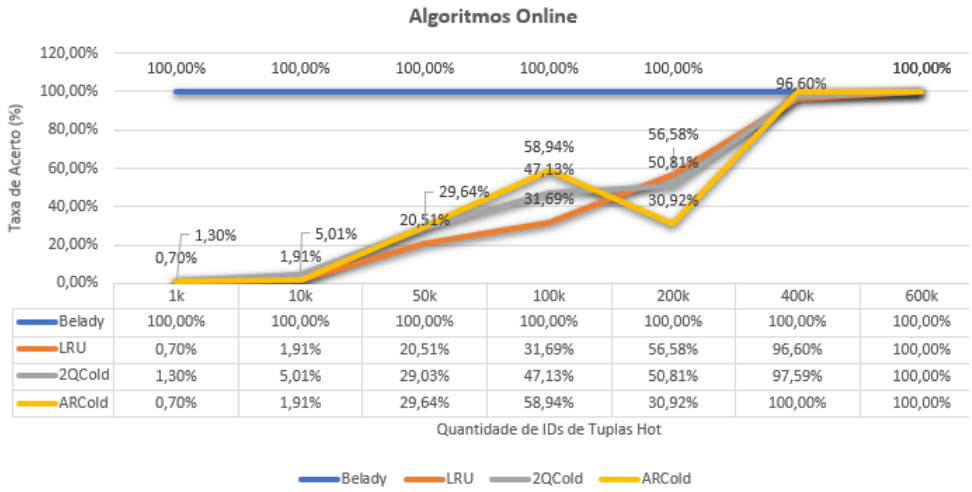


Figura 2. Tempo de Resposta em Milissegundos (Cenário Offline).

4.2.2. Taxa de Acerto

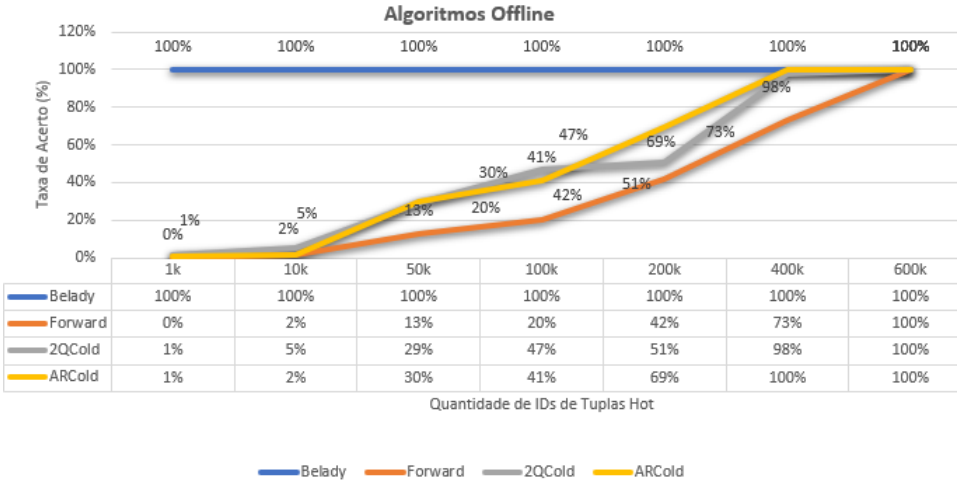
A Figura 3 ilustra as taxas de acerto obtidas pelos algoritmos *2QCold*, *ARCold*, *LRU* e *Belady* [Belady 1966] no cenário *offline*. Analisando a Figura 3 podemos observar que os algoritmos *2QCold* e *ARCold* obtiveram um aumento na taxa de acerto de cerca de 0,6% e 27,25%, respectivamente, em comparação com o algoritmo de *LRU*.

A Figura 4 ilustra as taxas de acerto obtidas pelos algoritmos *2QCold*, *ARCold*, *Forward* [Stoica et al. 2013] e *Belady* [Belady 1966] no cenário *offline*. Analisando a Figura 4 podemos observar que os algoritmos *2QCold* e *ARCold* obtiveram um aumento na taxa de acerto de cerca de 1% e 27%, respectivamente, em comparação com o algoritmo de *Forward* [Stoica et al. 2013].



	1k	10k	50k	100k	200k	400k	600k
Belady	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
LRU	0,70%	1,91%	20,51%	31,69%	56,58%	96,60%	100,00%
2QCold	1,30%	5,01%	29,03%	47,13%	50,81%	97,59%	100,00%
ARCold	0,70%	1,91%	29,64%	58,94%	30,92%	100,00%	100,00%

Figura 3. Taxa de Acerto (Cenário Online).



	1k	10k	50k	100k	200k	400k	600k
Belady	100%	100%	100%	100%	100%	100%	100%
Forward	0%	2%	13%	20%	42%	73%	100%
2QCold	1%	5%	29%	47%	51%	98%	100%
ARCold	1%	2%	30%	41%	69%	100%	100%

Figura 4. Taxa de Acerto (Cenário Offline).

5. Conclusão

Neste artigo, apresentamos dois novos algoritmos denominados *2QCold* e *ARCold*, que adaptam os algoritmos clássicos de cache *2Q* e *ARC* para identificar dados frios. Implementamos os algoritmos propostos utilizando *Seal-DB* e os comparamos com os algoritmos clássicos *LRU*, *Forward* [Stoica et al. 2013] e *Belady* [Belady 1966]. O *benchmark TPC-C* foi usado nos experimentos. Os resultados mostraram que os algoritmos *2QCold* e *ARCold* em suas versões *online* e *offline* reduziram o tempo de resposta do sistema em 5% e 28%, respectivamente. Além disso, ambos conseguiram aumentar a taxa de acerto na identificação dos dados frios em 27% em relação aos trabalhos relacionados.

A partir da identificação dos dados frios é possível implementar diversas estratégias para evitar o transbordamento de dados na memória. Por exemplo, os dados frios podem ser despejados em disco, compactados ou mesmo eliminados. Como trabalhos futuros, pretendemos implementar e avaliar diferentes abordagens para gerenciamento de dados frios. Além disso, iremos realizar novos experimentos utilizando os *benchmarks* TPC-C fator 10 e o TPC-H Fator 1.

Referências

Afify Ghada M, Bastawissy Ali El, H. O. M. (2016). Identifying hot / cold data in main-memory database using frequent item set mining. *International Journal of Enhanced Research in Management & Computer Applications*, pages 35–42.

Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101.

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

Eldawy, A., Levandoski, J., and Larson, P.-A. (2014). Trekking through siberia: Managing cold data in a memory-optimized database. In *Proceedings of the VLDB Endowment*, volume 7, pages 931–942.

Emmanuel, I. and Stanier, C. (2016). Defining big data. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, pages 1–6.

Funke Florian, Kemper Alfons, N. T. (2012). Compacting transactional data in hybrid oltp & olap databases. *Proceedings of the VLDB Endowment*, pages 1424–1435.

Ha, H., Shim, D., Lee, H., and Park, D. (2021). Dynamic hot data identification using a stack distance approximation. *IEEE Access*, 9:79889–79903.

Johnson Theodore, Shasha Dennis, N. B. O. W. G. (1994). 2q : A low overhead high performance management replacement algorithm. *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450.

Kemper Alfons, N. T. (2011). Hyper : A hybrid oltp & olap main memory database system based on virtual memory snapshots. *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206.

Lahiri, T., Neimat, M.-A., and Folkman, S. (2013). Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13.

Megiddo, N. and Modha, D. S. (2003). Arc: A self-tuning, low overhead replacement cache. *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, pages 115–130.

Moraes, G., Moraes Filho, J. d. A., and Brayner, A. (2017). Seal-db: Uma ferramenta de suporte ao aprendizado de banco de dados. *32th Brazilian Symposium on Databases DEMOS AND APPLICATIONS SESSION PROCEEDINGS*, pages 35–40.

Pathak, A., Gurajada, A., and Khadilkar, P. (2018). Life cycle of transactional data in in-memory databases. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 122–133. IEEE.

Stoica, R., Levandoski, J. J., and Larson, P.-A. (2013). Identifying hot and cold data in main-memory databases. *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, pages 26–37.