**Comparison of Database Buffer Management Algorithms – 2Q and CLOCK-Pro**

Ashish Walia

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

ashish.walia@nyu.edu

Dennis Shasha

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

shasha@cs.nyu.edu

June 04, 2012

1. **Introduction**

Database management systems (DBMSs) typically maintain a database buffer to provide efficient access to database pages [1]. A database buffer consists of page frames of the size of disk blocks [1]. Blocks are copied from disk into buffer without any change in format [1]. Since it’s more expensive to access a database page from disk than to access a page from the buffer [2], the problem space reduces to effectively managing pages in the buffer. Finding an efficient solution for database buffer management is a well-researched problem. Many algorithms have been proposed so far in the same context, such as LRU, LIRS, Clock-Pro, 2Q, CAR, ARC etc. In this report, our focus is restricted to comparing 2Q and CLOCK-Pro algorithms in terms of hit rate (ratio of number of pages requested that are already in the buffer to the total number of pages requested) over a series of real-time workloads. *I drew inspiration from [4] and [5] for writing this introduction but referred only [1] and [2] for the introduction. I can’t think of any more references. –Ashish [Ok, we want to avoid plagiarism, so be sure you have paraphrased rather than taking actual phrases]*

1. **Optimal Page Replacement Algorithm (OPT)**

The optimal page replacement algorithm or clairvoyant algorithm replaces a page whose next access will be farthest in future [3]. It’s impossible to implement OPT algorithm for practical purposes because the point when a page will be referenced next in the future is not known in advance [3]. However, OPT gives a natural best case among all the page replacement algorithms and can be used as a standard point of reference against which to compare performance of other page replacement algorithms. Our experiments were based on memory traces; hence we could easily look ahead in the future to see which page would be accessed farthest in the future. We used that information in our OPT algorithm implementation.

On accessing a page X :

begin

 if X is in the buffer then

 increment page hit counter

else // X not in the buffer

 increment page miss counter

 if buffer is not full then

 add X to the buffer

else // buffer is full

//Select a page that will not be referenced in the future for the / //longest time

page Y:=selectPageForEviction()

remove Y from the buffer

 end if

 end if

end

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

selectPageForEviction()

begin

 create a sub list, SL, of all the pages that have not been accessed so far

 //iterate over all the pages present in buffer

for i in 1..buffer.length loop

 page X := buffer(i)

 //calculate when a page will be accessed next time in the future

X.next\_reference\_time:=SL.indexOf(X.pageNum)

//if the page is not referenced in the future

if X.next\_reference\_time==-1 then

 //this page is the ideal candidate for page eviction

 return X

end if

 end loop

 //page with the highest value of next\_reference\_time will be selected for eviction

 page Y:= buffer(1)

for i in 1..buffer.length loop

 page X := buffer(i)

 if X. next\_reference\_time > Y. next\_reference\_time then

 Y:=X

 end if

 end loop

 return Y

end

1. **Two Queue Page Replacement Algorithm (2Q)**

2Q maintains two families of queues: hot and cold. When a page is referenced for the first time, 2Q places it in the “cold” A1 queue, managed as FIFO (first-in first-out) queue [4]. If the page is accessed again while in the A1 queue, then it’s probably a hot page and is promoted to “hot” Am queue, a queue managed as a LRU (least-recently-used) queue [4]. If the page is not accessed while in the cold queue, then it’s eventually removed from the cold queue [4]. 2Q deals with the problem of correlated references by further dividing the A1 queue into A1in (of maximum size Kin) and A1out (of maximum size Kout) queues [4]. The A1in queue keeps track of newly referenced pages whereas A1out queue keeps track of pages that have high long-term access rates [4]. Kin and Kout parameters are tuning parameters and set to fixed percentages of the cache size In our experiments Kin should range between 20% and 25% whereas Kout ranges between 60% and 65%, with the rest of memory being devoted to Am. [Ashish, is that the best way to state this or should we say “In our experiments, Kin = 20% and Kout = 65% works well as does Kin = 25% and Kout = 60% with the rest of the memory devoted to Am.]

**Pseudo code of 2Q algorithm as described in [4]:**

On Accessing a Page X:

begin

        if X is in Am then

            Move X to the head of Am

        else if X is in A1out then

            reclaimfor(X)

                Add X to the head of Am

        else if X is in A1in
                //do nothing

        else

                  reclaimfor(X)

                  Add X to the head of A1in

        end if

end

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//If there is space, we give it to X.

//If there is no space, we free a page slot to make room for large page X

reclaimfor(page X)

begin

        if there are free page slots then

                   put X into a free page slot

        else if (|A1in|>= Kin)

                    page out the tail of A1in, call it Y

                    add identifier of Y to the head of A1out

                   if (|A1out|>Kout)

                    remove identifier of Z from the tail of A1out

                    end if

                    put X into the reclaimed page slot

        else

                    page out the tail of Am, call it Y

                   //do not put it on A1out, it has not been accessed for a while

                    put X into the reclaimed page slot

        end if

end

1. **CLOCK-Pro Page Replacement Algorithm**

In [5], S. Jiang, F. Chen and X. Zhang describes reuse distance as the period of time in terms of the number of other distinct pages accessed since its last access. CLOCK-Pro uses reuse distance to categorize a page as either a hot page or a cold page [5]. A page is categorized as a hot page if it has a small reuse distance or as a cold page if it has large reuse distance [5]. CLOCK-Pro algorithm maintains a single circular list where all hot and cold pages are placed in the order of their accesses [5]. Hot pages are placed at the tail of the list whereas cold pages are placed at the head of the list [5].

Once a cold page is added to the list, it’s assigned a test period so that it gets a fair chance to compete with other hot pages in the list [5]. The test period is set as the time before a cold page passes the end of the list which means that the cold pages can be removed from the list as soon as they reach the end of the list [5] [this is still not clear. There is the head of the circular list, the tail of the circular list, and then the “end of the list”. What is that in relationship to the head and tail or is there a separate list? It seems we need pseudo-code] If the cold page is re-accessed during its test period, it turns into a hot page [5] [and then what happens? Does the page move somewhere?]. However, if the cold page is not re-accessed during its test period, it is removed from the list [5]. More about CLOCK-Pro algorithm i.e. its data structure search for the victim page and adaptive version of the algorithm are explained in [5].

1. **Experiments**

**5.1. Simulation on file I/O traces**

The file I/O traces used in this section are same as used for evaluation of the CLOCK-Pro algorithm in [5]. Quoting from [5]:

* + 1. **cpp** is a GNU C compiler pre-processor trace and is a member of the probabilistic pattern group.
		2. **multi2** is a member of the mixed pattern group and is obtained by executing three workloads, namely, cs, cpp and postgres, together.
		3. **sprite** is a Sprite network file system trace which contains requests to a file server from client workstations for a two-day period. It’s a member of temporally-clustered pattern group.
		4. **glimpse** is a text information utility trace and is a member of the loop pattern group.

For workload cpp, the performance of 2Q and CLOCK-Pro are comparable (see Table 1). Clock-PRO performs significantly better than 2Q for workloads multi2 and glimpse (see Table 2 and Table 4). However, for the sprite workload, the hit ratio of the 2Q algorithm is higher than CLOCK-Pro algorithm (see Table 3).

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache Size** | **2Q (Kin = 25% Kout=65%)** | **CLOCK-Pro** | **OPT** |
| 100 | .790 | .771 | .825 |
| 500 | .861 | .860 | .865 |
| 1000 | .865 | .864 | .865 |
| 2000 | .865 | .865 | .865 |
| 3000 | .865 | .865 | .865 |
| 4000 | .865 | .865 | .865 |
| 5000 | .865 | .865 | .865 |

**Table 1:** Hit Rate of 2Q, CLOCK-Pro and OPT algorithms on workload cpp.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache Size** | **2Q (Kin = 25% Kout=65%)** | **CLOCK-Pro** | **OPT** |
| 100 | .259 | .266 | .354 |
| 500 | .392 | .495 | .536 |
| 1000 | .506 | .567 | .621 |
| 2000 | .688 | .702 | .746 |
| 3000 | .767 | .778 | .784 |
| 4000 | .784 | .782 | .784 |
| 5000 | .784 | .784 | .784 |
| 6000 | .784 | .784 | .784 |
| 7000 | .784 | .784 | .784 |
| 8000 | .784 | .784 | .784 |
| 10000 | .784 | .784 | .784 |
| 20000 | .784 | .784 | .784 |

**Table 2:** Hit Rate of 2Q, CLOCK-Pro and OPT algorithms on workload multi2.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache Size** | **2Q (Kin = 25% Kout=65%)** | **CLOCK-Pro** | **OPT** |
| 100 | .366 | .280 | .508 |
| 500 | .857 | .768 | .879 |
| 1000 | .924 | .884 | .932 |
| 2000 | .942 | .923 | .948 |
| 3000 | .946 | .932 | .947 |
| 4000 | .947 | .937 | .947 |
| 5000 | .947 | .941 | .947 |
| 6000 | .947 | .943 | .947 |
| 7000 | .947 | .947 | .947 |
| 8000 | .947 | .947 | .947 |
| 10000 | .947 | .947 | .947 |
| 20000 | .947 | .947 | .947 |

**Table 3:** Hit Rate of 2Q, CLOCK-Pro and OPT algorithms on workload sprite.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache Size** | **2Q (Kin = 25% Kout=65%)** | **CLOCK-Pro** | **OPT** |
| 100 | .009 | .058 | .077 |
| 500 | .012 | .319 | .343 |
| 1000 | .461 | .501 | .531 |
| 2000 | .579 | .580 | .579 |
| 3000 | .579 | .580 | .579 |
| 4000 | .579 | .580 | .579 |
| 5000 | .579 | .580 | .579 |

**Table 4:** Hit Rate of 2Q, CLOCK-Pro and OPT algorithms on workload glimpse.

**5.2 Simulation on an Online Transaction Processing (OLTP) workload**

The I/O traces used in this section are obtained from [5]. These traces are from OLTP applications running at two large financial institutions as mentioned in [5]. We had to tweak the original implementation of CLOCK-PRO algorithm provided by Dr. Song Jiang, in order to execute it successfully on Finanical1.spc. We had experienced segmentation fault error upon running CLOCK-Pro algorithm on the Financial.spc trace and had to change the page table implementation from array to map in the original source code. We recorded a slightly better performance of the 2Q algorithm compared with the CLOCK-Pro algorithm for OLTP workload (see Table 5 and Table 6).

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache Size** | **2Q (Kin = 30% Kout=60%)** | **CLOCK-Pro** | **OPT** |
|  |  |  |  |
|  |  |  |  |
| 500000 | .867 | .839 | .867 |
| 1000000 | .867 | .867 | .867 |
| 2000000 | .867 | .867 | .867 |
| 3000000 | .867 | .867 | .867 |

**Table 5:** Hit Rate of 2Q, CLOCK-Pro and OPT algorithms on workload Financial1.spc.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cache Size** | **2Q (Kin = 30% Kout=60%)** | **CLOCK-Pro** | **OPT** |
|  |  |  |  |
|  |  |  |  |
| 500000 | .920 | .920 | .920 |
| 1000000 | .920 | .920 | .920 |
| 2000000 | .920 | .920 | .920 |
| 3000000 | .920 | .920 | .920 |

**Table 6:** Hit Rate of 2Q, CLOCK-Pro and OPT algorithms on workload Financial2.spc.

1. **Sensitivity of Parameters**

**6.1 2Q**

Choosing a value for Kin and Kout parameter for 2Q algorithm is essentially a tuning task. We experimented with different values of Kin and Kout parameters (see Table 7, Table 8, Table 9 and Table 10) and noted that Kin = 20% and Kout=60% did reasonalbly well for cpp and glimpse workloads whereas Kin = 25% and Kout=65% of cache size did well for multi2and sprite traces.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cache Size** | **2Q** **(Kin = 30%** **Kout=60%)** | **2Q****(Kin = 20%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=65%)** | **2Q****(Kin = 30%** **Kout=65%)** |
| 100 | .783 | .795 | .789 | .790 | .785 |
| 500 | .861 | .861 | .861 | .861 | .861 |
| 1000 | .865 | .865 | .865 | .865 | .865 |
| 2000 | .865 | .865 | .865 | .865 | .865 |
| 3000 | .865 | .865 | .865 | .865 | .865 |
| 4000 | .865 | .865 | .865 | .865 | .865 |
| 5000 | .865 | .865 | .865 | .865 | .865 |

**Table 7:** Hit Rate of 2Q algorithm with different values of Kin and Kout on cpp workload.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cache Size** | **2Q** **(Kin = 30%** **Kout=60%)** | **2Q****(Kin = 20%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=65%)** | **2Q****(Kin = 30%** **Kout=65%)** |
| 100 | .252 | .257 | .254 | .259 | .259 |
| 500 | .391 | .391 | .391 | .392 | .392 |
| 1000 | .506 | .506 | .506 | .506 | .506 |
| 2000 | .678 | .700 | .688 | .688 | .678 |
| 3000 | .761 | .773 | .767 | .766 | .760 |
| 4000 | .784 | .784 | .784 | .784 | .784 |
| 5000 | .784 | .784 | .784 | .784 | .784 |
| 6000 | .784 | .784 | .784 | .784 | .784 |
| 7000 | .784 | .784 | .784 | .784 | .784 |
| 8000 | .784 | .784 | .784 | .784 | .784 |
| 10000 | .784 | .784 | .784 | .784 | .784 |
| 20000 | .784 | .784 | .784 | .784 | .784 |

**Table 8:** Hit Rate of 2Q algorithm with different values of Kin and Kout parameters on multi2 workload.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cache Size** | **2Q** **(Kin = 30%** **Kout=60%)** | **2Q****(Kin = 20%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=65%)** | **2Q****(Kin = 30%** **Kout=65%)** |
| 100 | .357 | .364 | .361 | .366 | .363 |
| 500 | .854 | .857 | .856 | .857 | .855 |
| 1000 | .922 | .924 | .923 | .924 | .924 |
| 2000 | .942 | .942 | .942 | .942 | .942 |
| 3000 | .945 | .945 | .944 | .946 | .946 |
| 4000 | .947 | .947 | .947 | .947 | .947 |
| 5000 | .947 | .947 | .947 | .947 | .947 |
| 6000 | .947 | .947 | .947 | .947 | .947 |
| 7000 | .947 | .947 | .947 | .947 | .947 |
| 8000 | .947 | .947 | .947 | .947 | .947 |
| 10000 | .947 | .947 | .947 | .947 | .947 |
| 20000 | .947 | .947 | .947 | .947 | .947 |

**Table 9:** Hit Rate of 2Q algorithm with different values of Kin and Kout parameters on sprite workload.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cache Size** | **2Q** **(Kin = 30%** **Kout=60%)** | **2Q****(Kin = 20%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=60%)** | **2Q****(Kin = 25%** **Kout=65%)** | **2Q****(Kin = 30%** **Kout=65%)** |
| 100 | .009 | .009 | .009 | .009 | .009 |
| 500 | .012 | .012 | .012 | .012 | .012 |
| 1000 | .444 | .477 | .460 | .460 | .444 |
| 2000 | .579 | .579 | .579 | .579 | .579 |
| 3000 | .579 | .579 | .579 | .579 | .579 |
| 4000 | .579 | .579 | .579 | .579 | .579 |
| 5000 | .579 | .579 | .579 | .579 | .579 |

**Table 10:** Hit Rate of 2Q algorithm with different values of Kin and Kout parameters on glimpse workload.

*Kin = 20% and Kout=60% did reasonalbly well for cpp and glimpse workloads whereas Kin = 25% and Kout=65% of cache size did well for multi2and sprite traces.--Ashish*

[Could you see whether there is one setting for Kin/Kout for all workloads that worked well?]

**6.2 CLOCK-Pro**

CLOCK-Pro adapts to the different workloads and doesn’t require predetermined parameters [5]. Do you mean the test period? The test period is not a predetermined parameter. Quoting from [5]

*``cold/hot'' are defined based on relativity and by constant comparison in one clock, not on a fixed threshold that are used to separate the pages into two lists. This makes CLOCK-Pro distinctive from prior work including 2Q and CAR, which attempt to use a constant threshold to distinguish the two types of pages, and to treat them differently in their respective lists (2Q has two queues, and CAR has two clocks), which unfortunately causes these algorithms to share some of LRU's performance weakness.*

[What about its cold time?]

1. **Conclusion**

CLOCK-Pro and 2Q give comparable performance in all cases of interest. The differences are rarely more than 1 or 2%. Based on these experiments, we would recommend the use of whichever algorithm is easier to implement.

1. **Acknowledgements**

Many thanks to Dr. Song Jiang for providing us source code for CLOCK-Pro algorithm and memory traces used in [5]. We are also thankful to Laboratory of Advanced Systems Software, University of Massachusetts, Amherst, for making OLTP traces available on the web [6] courtesy of Ken Bates from HP, Bruce McNutt from IBM and the Storage Performance Council.

1. **References**

[1] J. M. Hellerstein, M. Stonebraker and J. Hamilton. Architecture of Database System.

 [2] W. Effelsberg and T. Haerder. Principles of Database Buffer Management.

 [3] OPT, <http://en.wikipedia.org/wiki/Page_replacement_algorithm>

[4] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer

 Management Replacement Algorithm. In Proc. of VLDB’94, 1994, pp 439-450.

[5] S. Jiang, F. Chen and X. Zhang. CLOCK-Pro: An Effective Improvement of the Clock

 Replacement. In Proc. of USENIX’05, April 2005.

[6] UMASSTraceRepository, <http://traces.cs.umass.edu/index.php/Storage/Storage>