2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm *

Theodore Johnson
University of Florida
Gainesville, FL 32611

Dennis Shasha
Courant Institute, New York University
New York, NY 10012
Novell, Inc.
Summit, NJ 07901

Abstract

In a path-breaking paper last year Pat and Betty O'Neil and Gerhard Weikum proposed a self-tuning improvement to the Least Recently Used (LRU) buffer management algorithm[15]. Their improvement is called LRU/k and advocates giving priority to buffer pages based on the kth most recent access. (The standard LRU algorithm is denoted LRU/1 according to this terminology.) If P1's kth most recent access is more more recent than P2's, then P1 will be replaced after P2. Intuitively, LRU/k for $k > 1$ is a good strategy, because it gives low priority to pages that have been scanned or to pages that belong to a big randomly accessed file (e.g., the account file in TPC/A). They found that LRU/2 achieves most of the advantage of their method.

The one problem of LRU/2 is the processor overhead to implement it. In contrast to LRU, each page access requires log N work to manipulate a priority queue where N is the number of pages in the buffer.

Question: is there low overhead way (constant overhead per access as in LRU) to achieve similar page replacement performance to LRU/2?

Answer: Yes.

Our "Two Queue" algorithm (hereafter 2Q) has constant time overhead, performs as well as LRU/2, and requires no tuning. These results hold for real (DB2 commercial, Swiss bank) traces as well as simulated ones. Based on these experiments, we estimate that 2Q will provide a few percent improvement over LRU without increasing the overhead by more than a constant additive factor.

*Supported by U.S. Office of Naval Research #N00014-91-J-1472 and #N00014-92-J-1719, U.S. National Science Foundation grants #CCR-9103953 and IRI-9224601, and USRA #5555-19. Part of this work was performed while Theodore Johnson was a 1993 ASEE Summer Faculty Fellow at the National Space Science Data Center of NASA Goddard Space Flight Center.

† Authors' e-mail addresses : ted@cis.ufl.edu and shasha@cs.nyu.edu

0

1 Background

Fetching data from disk requires at least a factor of 1000 more time than fetching data from a RAM buffer. For this reason, good use of the buffer can significantly improve the throughput and response time of any data-intensive system.

Until the early 80's, the least recently used buffer replacement algorithm (replace the page that was least recently accessed or used) was the algorithm of choice in nearly all cases. Indeed, the theoretical community blessed it by showing that LRU never replaces more than a factor B as many elements as an optimal clairvoyant algorithm (where B is the size of the buffer) [19]. [1]

Factors this large can heavily influence the behavior of a database system, however. Furthermore, database systems usually have access patterns in which LRU performs poorly, as noted by Stonebraker [21], Sacco and Schkolnick [18], and Chou and Dewitt [5]. As a result, there has been considerable interest in buffer management algorithms that perform well in a database system.

Much of the work in database buffer management

[1] A companion result is that LRU on a buffer of size B will never page more than twice as much as a clairvoyant algorithm on a buffer of size B/2.

has tuned the buffer management algorithm to the expected reference pattern. Reiter proposed the Domain Separation algorithm [16], which separates the database pages into different pools. Each pool has an allocation of buffers and a special purpose buffer management strategy. The allocation of buffers to pools requires careful tuning for high performance [7, 22] and is a well known headache for system administrators of database management systems such as DB2.

A related approach is to let the query optimizer tell the buffer manager the plan of the query to be processed, so that the buffer manager can allocate and manage its buffers accordingly. Algorithms include the Hot Set model [18], the DBMIN algorithm [5] and related extensions [9, 13, 23], and hint passing algorithms [4, 12, 1, 10, 3].

In [15], O'Neil, O'Neil and Weikum point out that it is difficult to design a query plan-based buffering algorithm that works well in a multitasking system. Sequences of transactions, concurrent transactions, and partions of the current transaction plan can overlap in complicated ways. In addition, buffer partitioning schemes reduce the effective available space, say, when one buffer is not fully utilized. New data access methods must use existing buffer management algorithms (which might not fit well), or have a new special-purpose algorithm written.

The O'Neil-Weikum paper provides an alternative to buffer partitioning. They suggested a more-or-less self-tuning full buffer variant of LRU with many of the same virtues as the hybrid schemes. The essence of their scheme is that a page should not obtain buffer privileges merely because it is accessed once. What's important is whether it has been popular over time. The version they advocate — called LRU/2 — replaces the page whose **penultimate** (second-to-last) access is least recent among all penultimate accesses.

In order to be able to find that page quickly, they organize pointers to the buffer pages in a priority queue based on penultimate access time. The operations are deletemin to get the page to replace and a promotion scheme to move a page that has been accessed to a new place in the queue. Priority queue operations entail a time complexity that grows with the log of the size of the buffer.

They examine (i) a synthetic workload with references to two pools of pages one for an index and one for data pages; (ii) a synthetic workload with random references based on a Zipfian distribution of reference frequencies obeying the 80-20 rule on 1000 possible pages; and (iii) a commercial on-line transaction processing workload derived from a Swiss bank. In every case, they show a significant improvement over LRU, sometimes as high as 40%.

While LRU/2 is self-tuning in comparison to other buffer management algorithms, such as Gclock [20, 8, 14], two delicate tuning parameters remain. The first is the *Correlated Reference Period.* This is a time period during which a page is retained in the buffer once it has been accessed. In this way close-in-time subsequent accesses from the same process (so-called "correlated" accesses) find the page in the buffer. At the same time, multiple accesses within a correlation period count only as a single access from the point of view of replacement priority. The net effect is that if a page is referenced several times within a correlated reference period and then isn't accessed for a while, it will have low buffer priority.

The second tuning parameter is the *Retained Information Period* which is the length of time a page's access history is remembered after it is ejected from the buffer. The authors recommend 200 seconds though higher values don't hurt.

A related paper is the frequency-based replacement algorithm of Robinson and Devarakonda [17]. These authors found that by not counting correlated references, they could produce a buffering algorithm based on reference frequency counting that has better performance than LRU.

Our contribution in this work is to provide a buffer management algorithm that is as good as the LRU/2 algorithm, but which is simpler, requires no tuning, and is much faster to execute (constant time rather than logarithmic). Both LRU/2 and 2Q are "scan resistant" and effectively handle index accesses (as our results section shows), so they can be used in a database system without requiring hints from the database manager or the query planner.

We note that 2Q can be used in conjunction with algorithms that use hints from the query planner, however, such as DBMIN. Work exploring combinations of DBMIN and LRU/k or DBMIN and 2Q are on our future research agenda.

2 2Q Algorithm Description

LRU works well because it tends to remove cold pages from the buffer to make space for the faulted page. If the faulting page is cold, however, then LRU may displace a warmer page to make space for the cold one. Furthermore, the cold page will reside in the buffer for a considerable amount of time. LRU/2 improves on LRU by using a more aggressive exit rule, thereby quickly removing cold pages from the buffer. Implementing the exit rule requires priority queue manipulations.

The ways in which 2Q and LRU/2 improve upon LRU are in some sense complementary. Instead of cleaning cold pages from the main buffer, 2Q admits only hot pages to the main buffer. As with LRU/2, 2Q tests pages for their second reference time. Simplifying slightly, on the first reference to a page, 2Q places it in a special buffer, the A1 queue, which is managed as a FIFO queue. If the page is re-referenced during its A1 residency, then it is probably a hot page. So, if a page in A1 is referenced, the page is moved to the Am queue, which is managed as an LRU queue. If a page is not referenced while on A1, it is probably a cold page, so 2Q removes it from the buffer.

Simplified 2Q

```
if p is on the Am queue
then
      put p on the front of the Am queue
      /* Am is managed as an LRU queue*/
else if p is on the A1 queue
then
      remove p from the A1 queue
      put p on the front of the Am queue
else /* first access we know about concerning p */
      /* find a free page slot for p */
      if there are free page slots available
      then
           put p in a free page slot
      else if A1's size is above a (tunable) threshold
           delete from the tail of A1
           put p in the freed page slot
      else
           delete from the tail of Am
           put p in the freed page slot
      end if
      put p on the front of the A1 queue
end if
```

When we experimented with the above algorithm, we found that tuning it was difficult. If the maximum size of A1 is set too small, the test for hotness is too strong and the buffer will take too long to load in the hot data items. If the maximum size of A1 is set too large, then A1 steals page slots from Am, and performance degrades because the buffer for the hot pages is small.

A solution to the tuning problems is to store pointers to the pages in A1 instead of the pages themselves. This solution lets A1 remember enough past references to be responsive to moderately hot pages, yet preserve all page slots for the Am queue. We found that this solution works well for traces generated from stationary distributions, but performs poorly for actual traces. In real access patterns, the locality of reference can change very quickly. A common occurrence is for a page to receive many references for a short period of time (i.e., correlated references), then no references for a long time. When a new page is accessed, it should be retained for a period of time to satisfy the correlated references, then ejected from the buffer if it does not receive any further references

2Q, Full Version

To resolve the above problems we partition A1 into A1in and A1out. The most recent first accesses will be in the memory (up to a certain threshold), but then older first accesses will be thrown out of memory but remembered in A1out. (Remembering a page address requires only a few bytes (fewer than 10).) Given a buffer size B, one divides it into Am, A1in (of maximum size Kin and minimum size 1), and A1out (of maximum size Kout). Fixing these parameters is potentially a tuning question though the following values work well: Kin should be 25% of the page slots and Kout should hold identifiers for as many pages as would fit on 50% of the buffer.

This algorithm solves the correlated reference problem by retaining newly referenced pages in A1in to satisfy repeated requests. If a page in A1in is accessed, it isn't promoted to Am, because the second access may simply be a correlated reference. The A1out buffer, by contrast, is used to detect pages that have high long-term access rates.

```
// If there is space, we give it to X.
// If there is no space, we free a page slot to
// make room for 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 hasn't been
        // accessed for a while
        put X into the reclaimed page slot
    end if
end
```

```
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        // X is in no queue
        reclaimfor(X)
        add X to the head of A1in
    end if
end
```

3 Experimental Results

In this section, we show that 2Q is better than LRU and Gclock, and is comparable to LRU/2. In all our experiments, we tried to represent LRU/2 as fairly as possible, but it was difficult to model the tunables of that algorithm exactly.

The LRU/2 algorithm keeps a page in the buffer for a grace period (the Correlated Reference Period) and considers a second access to be a legitimate second access only if it arrived after that period. It also remembers pages that have been ejected for 200 seconds. To implement the LRU/2 algorithm with a correlated reference period, we used the A1in queue. On its first access, a page is added to the A1in queue. The page it displaces is inserted into the LRU/2 priority queue, and a deletemin operation (on the penultimate access time) is run to determine a page to eject from the buffer (perhaps the one just inserted from the A1in queue). References to a page in the A1in queue are not counted. We adjusted the size of the A1in queue to obtain the best performance for the LRU/2 algorithm.

We implemented the Gclock algorithm as described in [14]: every page has a history count attached. When a page is referenced, we set the history count of its buffer to *init_count*. If a free page is required, we scan through the pages (starting from the previous stopping point) and examine the history counts. If the history count of a page is non-zero, then we decrement the history count and move on to the next page. Otherwise, we choose that page for replacement. We tested Gclock with an *init_count* parameter set to 2 and 4, and report the best results. Gclock usually has a slightly lower hit rate than LRU, and is never higher than 2Q, so we report Gclock results in the tables only, since it is not a serious contender.

We also tested the Second Chance algorithm, which is similar to Gclock but is designed for a virtual memory system. When a page is referenced, its *reference bit* is set. During a scan for replacement: if the reference bit of a page is set then its reference bit is cleared and its history bit is set; if its history bit is set, the bit is reset; if neither its reference bit nor its history bit is set, the page is selected for replacement.

3.1 Simulation Experiments

The experiments in this subsection are from artificially generated traces. They give some intuition about the behavior of LRU, LRU/2, and 2Q, but do not necessarily represent a real application's behavior.

Zipf Distributions

Figures 1 and 2 compare the performance of LRU, LRU/2, and 2Q on a Zipfian input distribution [11] with parameter $\alpha = 0.5$ and $\alpha = 0.86$ respectively. That is, if there are N pages, the probability of accessing a page numbered i or less is $(i/N)^\alpha$. A setting of $\alpha = 0.86$ gives an 80/20 distribution, while a setting of $\alpha = .5$ give a less skewed distribution (about 45/20). These obey the independent reference model[2], i.e., each page has a single stationary probability to be accessed at any point in time. We set Kin to 1 (since there are no correlated references) and Kout to 50% of the number of page slots in the buffer. We run the simulator for one million references, and report the

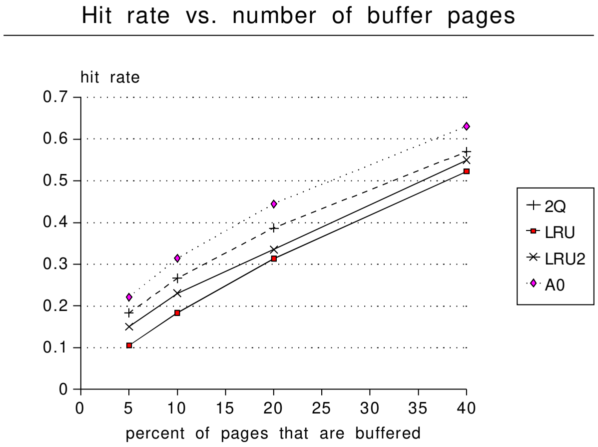

Figure 1: Zipf input comparison with parameter 0.5

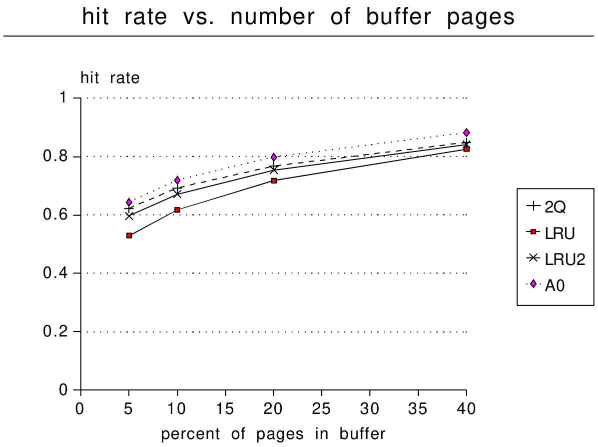

Figure 2: Zipf input comparison with parameter 0.8

number of hits divided by one million. We explicitly include the effects of the startup transient to provide a fair comparison, since both 2Q and LRU/2 sacrifice responsiveness to achieve a higher long-term hit rate. We simulated a database of 50,000 pages and buffer sizes ranging from 2,500 page slots to 20,000 page slots.

As a reference point, we included the hit rate of the A0 algorithm (A0 always replaces the page whose probability of access is lowest) which is known to be optimal for stable probability distributions[6]. These charts show that 2Q provides a substantial improvement over LRU, and has performance comparable to that of LRU/2. The performance improvement is highest when the number of page slots is small.

Mixed Zipf/Scans

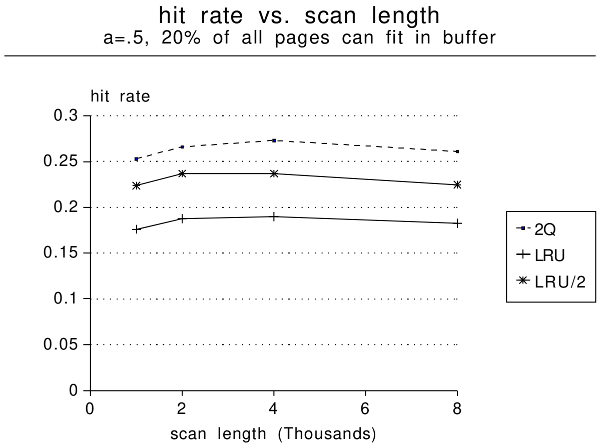

Figure 3: 1/3 Scan Input Mixed with Zipf Input Having Parameter 0.5

Both the LRU/2 and the 2Q algorithms are designed to give low priority to scanned pages. We modified the Zipf simulator so that it would occasionally start a scan. When simulating a scan, no page in the scan is repeated. We adjusted the probability of starting a scan so that one third of the references generated are due to scans.

We ran two experiments, each of which had a database size of 50,000. In the first experiment (Figure 3), we used a Zipf distribution with parameter $\alpha = .5$ and 10,000 page slots. In the second experiment (Figure 4) we used $\alpha = .86$ and 5,000 page slots. In both experiments we varied the scan length and reported the hit rates of LRU, LRU/2, and 2Q (with |A1in|=1 and |A1out|=50% of the number of page slots).

The experiments show that 2Q and LRU/2 have a considerably higher hit rate than LRU. The hit rates are lower than in the no-scan case, however. This is to be expected because there is little hope of getting a hit on a scan. However, we notice that the the hit rate of 2Q suffers less than the hit rate of LRU. The hit rate of 2Q is about 70% of its no-scan hit rate, while the hit rate of LRU is about 60% of its no-scan hit rate. Thus, 2Q provides a kind of "scan resistance."

Online Transaction Processing-style Index Accesses

A nice observation of the LRU/2 paper is that randomly accessed data pages get an excessively high priority under LRU. The authors posited an example where there are 100 index pages and 10,000 data pages. References come in pairs, one page selected uniformly at random from the set of index pages and

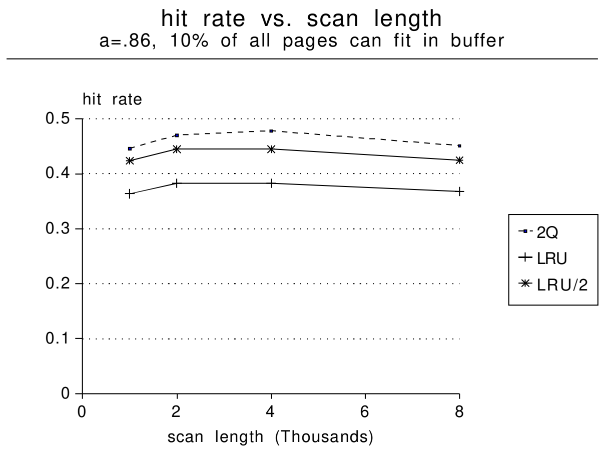

Figure 4: 1/3 Scan Input Mixed with Zipf Input Having Parameter 0.8

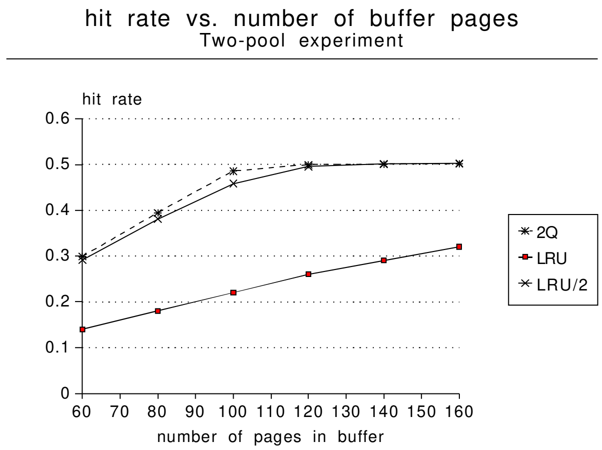

Figure 5: Behavior of Algorithms on Index Accesses to Data Pages

one selected uniformly at random from the set of data pages. The number of available page slots varies from less than 100 to greater than 100. They showed that LRU/2 gives priority to the index pages as it should, since their hit rate with 100 page slots is almost 1/2. We ran a simulation with identical parameters, and we found that 2Q also gives preference to index pages (see Figure 5).

3.2 Experiments on Real Data

Colleagues at IBM and Novell (UNIX Systems Group) kindly supplied traces of a commercial DB2 database application and of hits on program text for a windowing graphics application, respectively. We ran experiments using the same parameters for these two (very different) applications. 2Q performed well on both.

For these experiments we set Kout to 50% of the total number of page slots (we arrived at this setting based on our simulation studies, described in the next section). We set Kin to 20% and 30% of the total number of page slots to test the sensitivity of the 2Q algorithm to the Kin parameter. Since the differences among the various algorithms sometimes fall below 5% we represent these results with tables.

Trace Data from a DB2 Commercial Application

The DB2 trace data came from a commercial installation of DB2. We received a trace file containing 500,000 references to 75514 unique pages. The file was fed to our buffer management simulators. The results in Table 1 show that both 2Q and LRU/2 provide a significantly higher hit rate than LRU and Gclock.

The best performance is obtained when the size of Kin is matched to the amount of correlated reference activity, though the sensitivity is slight enough to be ignored.

Trace Data from Program Text Accesses of a Windowing Application

Colleagues at the UNIX Systems Group of Novell supplied us with a trace of program references made on a windowing application. We received a trace file containing 427,618 references to 290 256 byte pages. The results in Table 2 show again that both LRU/2 and 2Q provide a significantly higher hit rate than LRU and Gclock.

Trace Data from an On-line Transaction Processing System

Gerhard Weikum supplied us with the trace he used to validate the performance of the LRU/2 algorithm [15], a one hour page reference trace to a CODASYL database. The trace consists of of 914145 references to 186880 distinct pages. The results in Table 3 further validate our results. We note that our results on this trace are different than those reported in [15] because the LRU/k authors removed some references in their preprocessing. However, our conclusions about relative performance are the same.

Rerunning the Synthetic Experiments with Default Tuning Parameters

Since 2Q with a single set of parameters works so well with trace files from very different sources and with very different characteristics, we conjectured that our suggested parameter settings are almost always good and 2Q needs little tuning. To confirm our conjecture, we ran the simulations with a Zipf distribution

Number of page slots	LRU/2	LRU	Gclock	2nd Chance	2Q Kin=30%	2Q Kin=20%
50	.345	.317	.313	.313	.349	.349
100	.432	.413	.407	.407	.444	.430
200	.516	.512	.506	.506	.521	.510
500	.610	.606	.602	.602	.624	.611
700	.639	.630	.627	.627	.658	.648
1000	.665	.654	.651	.651	.686	.681
1200	.678	.667	.664	.664	.698	.695
1500	.693	.683	.680	.680	.711	.712
1700	.702	.691	.688	.688	.719	.720
2000	.712	.704	.700	.700	.729	.730
2500	.727	.718	.715	.715	.742	.744
3000	.738	.730	.726	.726	.750	.753
3500	.747	.740	.736	.736	.759	.760
4000	.755	.748	.744	.744	.767	.766
5000	.768	.762	.758	.760	.778	.777
6000	.778	.773	.768	.768	.785	.786
7000	.786	.781	.777	.777	.791	.791
8000	.792	.789	.784	.784	.795	.795
9000	.798	.795	.790	.790	.798	.799
10000	.802	.801	.796	.796	.801	.801

Table 1: DB2 trace hit rate comparison of LRU, LRU/2, and 2Q.

with Kin set to 25% of the page slots and Kout set to 50% of the page slots. For a fair comparison, we used an A1in queue for the LRU/2 algorithm. The results are in Table 4. In spite of the fact that there are no correlated references in the reference string, both 2Q and LRU/2 show a substantial improvement over LRU and Gclock.

4 Setting the Parameters

Recall that the 2Q algorithm has a queue called Am for pages hit at least two and possibly more times, a queue A1in that stores pages that have been hit only once (in recent memory), and a queue A1out that stores pointers of pages that have been hit only once. To gain intuition about the sensitivity to various sizes, we performed the following experiments.

The Size of A1out

When we developed the Simplified 2Q algorithm, we ran experiments to determine if the pages in A1 should be stored, or if only the tags should be stored. If A1 holds pages, then a page in A1 will not cause a page fault if it is re-referenced, but will steal pages from Am. We ran experiments to see which approach (pages or tags) was better. A Zipf distribution on 50,000 data items was given to the 2Q algorithm. In the first experiment, the parameter of the Zipf distribution is set to $\alpha = .86$ and 10,000 page slots are used. In the second experiment, $\alpha = .5$ and 5,000 page slots are used. In both experiments, we made all A1 entries store either pages or tags. The results are in Figure 6. We found that the storing the A1 tags is is considerably better than storing the A1 pages. Further, the performance of 2Q algorithm is not sensitive to the size of A1 when A1 stores tags, while the algorithm is highly sensitive when A1 stores pages.

For this reason, the Full 2Q algorithm uses tags (i.e., A1out) to filter the pages that are admitted to Am. The A1in queue is used to handle correlated references only.

Responsiveness

The chart in the previous experiment suggests that the best setting of the size of A1out is about 5% of the number of page slots in Am. However, we found that such a small A1out queue did not give good performance on real input. The problem with using such a small A1out queue is that it admits pages too selectively into the Am queue, and cannot respond to changes in locality.

Number of page slots	LRU/2	LRU	Gclock	2nd Chance	2Q Kin=30%	2Q Kin=20%
4	.443	.428	.427	.427	.457	.457
6	.504	.479	.484	.484	.533	.532
8	.551	.536	.530	.530	.598	.598
10	.602	.578	.577	.577	.662	.661
12	.678	.655	.657	.657	.722	.721
14	.740	.713	.703	.703	.762	.765
16	.762	.730	.722	.722	.785	.792
18	.790	.759	.754	.753	.815	.816
20	.809	.772	.767	.767	.832	.836
22	.825	.795	.797	.797	.836	.844
24	.852	.825	.818	.818	.845	.849
26	.866	.839	.832	.832	.858	.866
28	.877	.849	.845	.845	.876	.879
30	.883	.861	.858	.858	.894	.896
32	.893	.876	.881	.881	.911	.913
34	.917	.909	.910	.910	.923	.926
36	.930	.925	.924	.924	.932	.934
38	.937	.935	.932	.932	.940	.941
40	.943	.946	.939	.939	.946	.946

Table 2: Windowing code trace hit rate comparison of LRU, LRU/2, and 2Q.

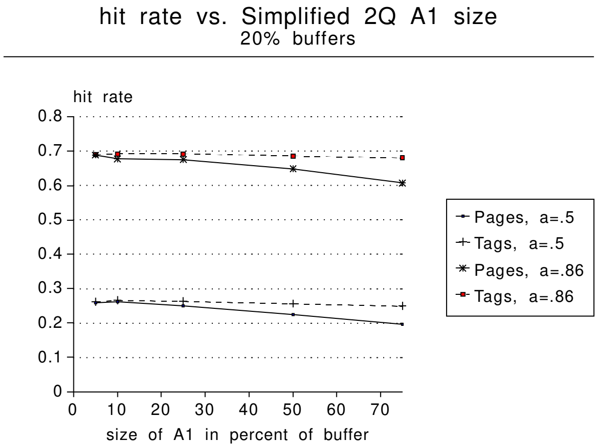

Figure 6: The Effect of the Size of A1in on the Hit Rate, assuming Zipf 0.5

To investigate the responsiveness of this algorithm to changes in locality, we ran the algorithms for 1,000,000 Zipf references, then made a random permutation of the probabilities assigned to the items (thus effecting a change in locality). Figures 7 and 8 show hit rates as a function of time after the permutation. The LRU algorithm recovers very quickly from a change in locality (which makes LRU harder to beat in practice than in theory). The LRU/2 algorithm algorithm recovers quickly, but not as fast as LRU. Thus, LRU/2 has sacrificed some responsiveness for a greater long-term hit rate. When the 2Q algorithm uses a small A1out (5% of the Am page slots), the responsiveness is very poor, and the hit rate still has not recovered after 1,000,000 references[2] When a moderately large A1out is used (50% of the number of page slots in A1in), 2Q is almost as responsive as LRU/2. Since the long-term performance suffers little when the size of A1out is increased from 5% to 50% but the responsiveness is greatly increased, we have used an A1out size of 50% in our performance comparisons.

[2]This counter-intuitive phenomena occurs because we tuned the 2Q algorithm to quickly fill up when empty. Flushing cold pages takes longer.

Number of page slots	LRU/2	LRU	Gclock	2nd Chance	2Q Kin=30%	2Q Kin=20%
100	.086	.083	.083	.083	.096	.090
200	.164	.144	.144	.141	.196	.181
500	.284	.234	.236	.223	.334	.329
1000	.384	.328	.327	.318	.405	.405
2000	.454	.425	.425	.418	.465	.464
5000	.544	.537	.538	.532	.556	.557
10000	.616	.607	.607	.602	.626	.624
20000	.678	.671	.671	.665	.681	.680

Table 3: OLTP trace hit rate comparison of LRU, LRU/2, and 2Q.

Number of page slots	LRU/2	LRU	Gclock	2Q
$\alpha = .5$				
5%	.141	.105	.104	.162
10%	.222	.183	.181	.238
20%	.347	.313	.311	.356
40%	.544	.529	.519	.535
$\alpha = .86$				
5%	.584	.528	.524	.595
10%	.661	.618	.613	.667
20%	.745	.718	.713	.744
40%	.838	.826	.821	.827

Table 4: Independent Zipf hit rate comparison of LRU, LRU/2, and 2Q.

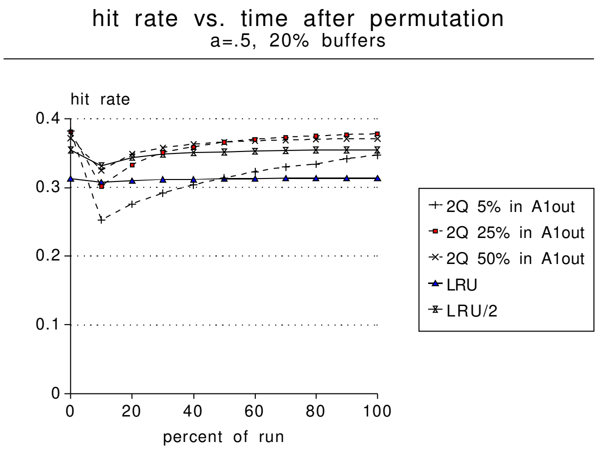

Figure 7: Responsiveness and the Size of A1, with Zipf 0.5

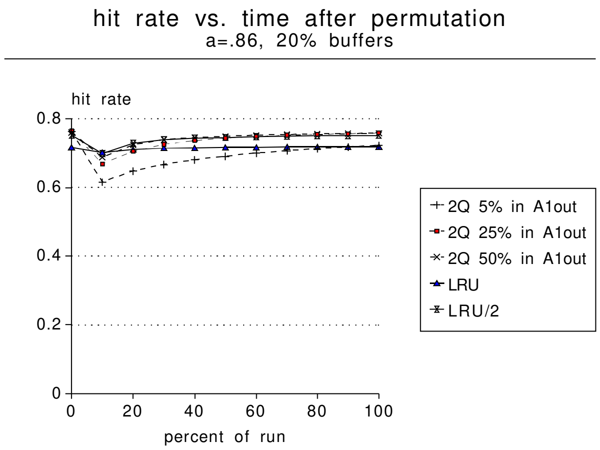

Figure 8: Responsiveness and the Size of A1, with Zipf 0.8

4.1 Theoretical Setting for 2Q Parameters

On a static reference stream, the A0 algorithm [6] is optimal. That algorithm effectively locks the B-1 data items with the highest access probability into memory, and use the last page slot to service page faults. LRU works well because it tends to keep hot items near the front of the queue, removing the cold ones. However, LRU is too trusting. It will admit a cold item into the buffer on the first hit and kick out a hotter item to make room. LRU/2 tries to avoid this problem by preferentially cleaning cold pages from the buffer. We would like to filter the items that are admitted to the Am queue, to ensure that we make room only for pages that are hot over the long term.

To help this goal, we could in principle look at the inter-reference times (coming from different threads of execution). If a data item has a short inter-reference time, we can feel confident that it is a hot item, and thus feel confident that we will not remove a hot page from the buffer to make room for a cold page. We have not done that in these traces, because it appeared not to be necessary.

In the Simplified 2Q algorithm, the A1 queue acts as a filter, as does the A1out queue in the Full 2Q algorithm. An item will be admitted to the Am queue only if it receives a second reference before being removed from the A1out queue. We would like the most popular pages to be in that queue. Suppose that the miss rate (the fraction of references that are misses using the A0 algorithm) is m, and there are f spaces available in the A1out area. We have observed that the A1out queue is usually full (the miss rate is much greater than the A1out hit rate). Thus, an item is admitted to the Am queue if it receives a second reference within f/m page references of entering A1out. Let us call this probability p_{accept}. Suppose that item i is referenced with probability p_i. The probability that item i gains admission to the Am queue on the k^{th} reference after entering A1out is $p_i(1-p_i)^{k-1}$. Therefore,

$$p_{accept} = \sum_{k=1}^{f/m} p_i(1-p_i)^{k-1} = 1-(1-p_i)^{f/m}$$

The probability of admission is close to one for large values of p_i, then drops off rapidly to zero when p_i becomes small. Thus, the A1out queue does indeed act as a filter, admitting hot items into the queue and ejecting cold items.

We can define the A1out filter's cutoff hotness as the reference probability, p_{cutoff}, at which $p_{accept} = 1/2$. The derivative of p_{accept} is large at this point, and an item with reference probability p_{cutoff} will be admitted to Am after three references instead of two.

So, we solve

$$1-(1-p_i)^{f/m} = 1/2$$

for the value of interest. Given f and m, we can ask for the access probability of the coldest item which has a 50% chance of being admitted to the Am queue. Then

$$\begin{aligned} p_{cutoff} &= 1-(1/2)^{m/f} \\ &\approx m\ln(2)/f \end{aligned}$$

(The approximation is by taking a Taylor series and has less than 4% error when $m/f \leq .1$).

We note that the A1out filter is self-adjusting. When the miss rate is high, p_{cutoff} is relatively high, so the queue will admit only the hottest items. When the miss rate is low, p_{cutoff} is relatively small, so that only the moderately hot items that should be admitted to the queue have a chance of gaining admission.

Next, given p_i and m, we can ask what $f = f_{crit}$ gives $p_{accept} = .5$. We find that

$$\begin{aligned} f_{crit} &= -\tfrac{m\ln(2)}{\ln(1-p_i)} \\ &\approx m\ln(2)/p_i \end{aligned}$$

The approximation follows because $\ln(1-p_i) \approx -p_i$. If we know the access probability of the coldest item that should be in the Am queue, we can solve for f. However, we typically do not know the access probabilities ahead of time (otherwise we'd use the *A0* algorithm). For a rough estimate of f, let us find a value of f, f_{approx}, that will admit an item of average hotness to the Am queue 50% of the time. If the miss rate is m and there are B page slots in Am, the average access probability for an item in the buffer is $(1-m)/B$. Substituting for p_i in the formula for f_{crit}, we find that

$$f_{approx} = \frac{\ln(2)mB}{1-m}$$

Since this formula requires an a priori estimate of the miss rate, it is of little use in practical tuning. However, it suggests that the size of A1out should be a fixed fraction of the size of the buffer, B. Our simulation experiments indicate that this fraction should be about 1/2 for values of m between 10% and 90%. For systems with a lower miss rate, the formula implies that a smaller A1out queue is better.

The number of page slots is usually a small fraction of the database size. Let D be the number of data items, and let $B = rD$. Then:

$$p_{cutoff} = 2m\ln(2)/(rD)$$

The average reference probability of a cold data item is $m/(1-r)D$. If we let $X = m/D$, then we find that the average cold data item has a reference probability of $\frac{1}{1-r}X$, and that:

$$p_{cutoff} = \frac{2\ln(2)}{r}X$$

Since r is small, $\frac{2\ln(2)}{r} \gg \frac{1}{(1-r)}$. Thus, setting $f = B/2$ filters out the average cold data item as long as r is small. Furthermore, $f_{approx}/B = 1/2$ when $m = 1/(1+2\ln(2)) \approx .419$. The value of f_{approx} does not vary rapidly with m as long as m is not large (if m is large, little can be done). We conclude that the performance of the algorithm is not sensitive to the exact setting of f, and that $f = B/2$ is almost always a good choice.

5 Conclusion

2Q is a good buffering algorithm (giving a 5-10% improvement in hit rate over LRU for a wide variety of applications and buffer sizes and never hurting), having constant time overhead, and requiring little or no tuning. It works well for the same intuitive reason that LRU/2 works well: it bases buffer priority on sustained popularity rather than on a single access.

2Q seems to behave as well as LRU/2 in our tests (slightly better usually, in fact) can be implemented in constant time using conventional list operations rather than in logarithmic time using a priority queue, and both analysis and experiment suggest it requires little or no tuning. Finally, it can potentially be combined with buffer hint-passing algorithms of the DBMIN family.

6 Acknowledgments

Many thanks to Ted Messinger of IBM and Steve Pendergrast of Novell and Gerhard Weikum for supplying us with real data. Thanks also to Gerhard Weikum, Pat and Betty O'Neil for friendly discussions and program code — would that all algorithmic competitions were so civil.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, 1990.

[2] D.I. Aven, E.G. Coffman, and Y.A. Kogan. *Stochastic Analysis of Computer Storage*. D. Reidel Publishing, 1987.

[3] C.Y. Chan, B.C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 444–454, 1992.

[4] E.E. Chang and R.H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented dbms. In *Proc. 1989 ACM SIGMOD Conf.*, pages 348–357, 1989.

[5] H.T. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. 11th ACM SIGMOD Conf.*, pages 127–141, 1985.

[6] E.G. Coffman and P.J. Denning. *Operating System Theory*. Prentice-Hall, 1973.

[7] A. Dan and D. Towsley. An approximate analysis of lru and fifo buffer replacement schemes. In *Proc. 1990 ACM SIGMETRICS Conf.*, pages 143–149, 1990.

[8] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(9):560–595, 1984.

[9] C. Faloustos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. 17th Conf. on Very Large Data Bases*, pages 265–274, 1991.

[10] R. Juahari, M. Carey, and M. Linvy. Priority hints: An algorithm fir priority-based buffer management. In *Proc. 16th Int'l Conf. on Very Large Data Bases*, 1990.

[11] D. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, 1973.

[12] L.A. Haas et al. Starburst midflight: As the dust clears. *IEEE Trans. on Knowledge and Database Systems*, 2(1):143–160, 1990.

[13] R. Ng, C. Faloustos, and T. Sellis. Flexible buffer management based on marginal gains. In *1991 ACM SIGMOD Conf.*, pages 387–396, 1991.

[14] V.F. Nicola, A. Dan, and D.M. Dias. Analysi of the generalized clock buffer replacement scheme for database transaction processing. In *Proc. 1992 ACM SIGMETRICS Conf.*, pages 35–46, 1992.

[15] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM Sigmod International Conference on Management of Data*, pages 297–306, 1993.

[16] R. Reiter. A study of buffer management policies for data management systems. Technical Report 1619, University of Wisconson (Madison) Mathematics Research Center, 1976.

[17] J.T. Robinson and M.V. Devarakonda. Data cache management using frequency-based replacement. In *ACM SIGMETRICS Conference*, pages 134–143, 1990.

[18] G.M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, 1986.

[19] D.D. Sleator and R.T. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[20] A.J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, 1978.

[21] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–428, 1981.

[22] J.Z. Teng and R.A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[23] P.S. Yu and D.W. Cornell. Optimal buffer allocation in a multi-query environment. In *Proc. 7th Int'l Conf. on Data Engineering*, pages 622–631, 1991.