The LRU-K Page Replacement Algorithm For Database Disk Buffering

Elizabeth J. O'Neil[1], Patrick E. O'Neil[1], Gerhard Weikum[2]

1 Department of Mathematics and Computer Science
University of Massachussetts at Boston
Harbor Campus
Boston, MA 02125-3393

2 Department of Computer Science
ETH Zurich
CH-8092 Zurich
Switzerland

E-mail: eoneil@cs.umb.edu, poneil@cs.umb.edu, weikum@inf.ethz.ch

ABSTRACT

This paper introduces a new approach to database disk buffering, called the LRU-K method. The basic idea of LRU-K is to keep track of the times of the last *K* references to popular database pages, using this information to statistically estimate the interarrival times of references on a page by page basis. Although the LRU-K approach performs optimal statistical inference under relatively standard assumptions, it is fairly simple and incurs little bookkeeping overhead. As we demonstrate with simulation experiments, the LRU-K algorithm surpasses conventional buffering algorithms in discriminating between frequently and infrequently referenced pages. In fact, LRU-K can approach the behavior of buffering algorithms in which page sets with known access frequencies are manually assigned to different buffer pools of specifically tuned sizes. Unlike such customized buffering algorithms however, the LRU-K method is self-tuning, and does not rely on external hints about workload characteristics. Furthermore, the LRU-K algorithm adapts in real time to changing patterns of access.

1. Introduction

1.1 Problem Statement

All database systems retain disk pages in memory buffers for a period of time after they have been read in from disk and accessed by a particular application. The purpose is to keep popular pages memory resident and reduce disk I/O. In their "Five Minute Rule", Gray and Putzolu pose the following tradeoff: We are willing to pay more for memory buffers up to a certain point, in order to reduce the cost of disk arms for a system ([GRAYPUT], see also [CKS]). The critical buffering decision arises when a new buffer slot is needed for a page about to be read in from disk, and all current buffers are in use: What current page should be dropped from buffer? This is known as the page replacement policy, and the different buffering algorithms take their names from the type of replacement policy they impose (see, for example, [COFFDENN], [EFFEHAER]).

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0297...$1.50

The algorithm utilized by almost all commercial systems is known as LRU, for Least Recently Used. When a new buffer is needed, the LRU policy drops the page from buffer that has not been accessed for the longest time. LRU buffering was developed originally for patterns of use in instruction logic (for example, [DENNING], [COFFDENN]), and does not always fit well into the database environment, as was noted also in [REITER], [STON], [SACSCH], and [CHOUDEW]. In fact, the LRU buffering algorithm has a problem which is addressed by the current paper: that it decides what page to drop from buffer based on too little information, limiting itself to only the time of last reference. Specifically, LRU is unable to differentiate between pages that have relatively frequent references and pages that have very infrequent references until the system has wasted a lot of resources keeping infrequently referenced pages in buffer for an extended period.

Example 1.1. Consider a multi-user database application, which references randomly chosen customer records through a clustered B-tree indexed key, CUST-ID, to retrieve desired information (cf. [TPC-A]). Assume simplistically that 20,000 customers exist, that a customer record is 2000 bytes in length, and that space needed for the B-tree index at the leaf level, free space included, is 20 bytes for each key entry. Then if disk pages contain 4000 bytes of usable space and can be packed full, we require 100 pages to hold the leaf level nodes of the B-tree index (there is a single B-tree root node), and 10,000 pages to hold the records. The pattern of reference to these pages (ignoring the B-tree root node) is clearly: I1, R1, I2, R2, I3, R3, . . ., alternate references to random index leaf pages and record pages. If we can only afford to buffer 101 pages in memory for this application, the B-tree root node is automatic; we should buffer *all* the B-tree leaf pages, since each of them is referenced with a probability of .005 (once in each 200 general page references), while it is clearly wasteful to displace one of these leaf pages with a data page, since data pages have only .00005 probability of reference (once in each 20,000 general page references). Using the LRU algorithm, however, the pages held in memory buffers will be the hundred most recently referenced ones. To a first approximation, this means 50 B-tree leaf pages and 50 record pages. Given that a page gets no extra credit for being referenced twice in the recent past and that this is more likely to happen with B-tree leaf pages, there will even be slightly *more* data

pages present in memory than leaf pages. This is clearly inappropriate behavior for a very common paradigm of disk access.□

Example 1.2. As a second scenario where LRU retains inappropriate pages in cache, consider a multi-process database application with good "locality" of shared page reference, so that 5000 buffered pages out of 1 million disk pages get 95% of the references by concurrent processes. Now if a few batch processes begin "sequential scans" through all pages of the database, the pages read in by the sequential scans will replace commonly referenced pages in buffer with pages unlikely to be referenced again. This is a common complaint in many commercial situations: that cache swamping by sequential scans causes interactive response time to deteriorate noticeably. Response time deteriorates because the pages read in by sequential scans use disk arms to replace pages usually held in buffer, leading to increased I/O for pages that usually remain resident, so that long I/O queues build up.□

To reiterate the problem we see in these two examples, LRU is unable to differentiate between pages that have relatively frequent reference and pages that have very infrequent reference. Once a page has been read in from disk, the LRU algorithm guarantees it a long buffer life, even if the page has never been referenced before. Solutions to this problem have been suggested in the literature. The previous approaches fall into the following two major categories.

- *Page Pool Tuning:*
  Reiter, in his Domain Separation algorithm [REITER], proposed that the DBA give better hints about page pools being accessed, separating them essentially into different buffer pools. Thus B-tree node pages would compete only against other node pages for buffers, data pages would compete only against other data pages, and the DBA could limit the amount of buffer space available for data pages if re-reference seemed unlikely. Such "pool tuning" capabilities are supported by some commercial database systems and are used in applications with high performance requirements (see, e.g., [TENGGUM, DANTOWS, SHASHA]). The problem with this approach is that it requires a great deal of human effort, and does not properly handle the problem of evolving patterns of hot-spot access (locality within the data page pool, changing over time).

- *Query Execution Plan Analysis:*
  Another suggestion was that the query optimizer should provide more information about the type of use envisioned by a query execution plan, so that the system will know if re-reference by the plan is likely and can act accordingly (see the Hot Set Model of [SACSCH], the DBMIN algorithm of [CHOUDEW]) and its extensions [FNS, NFS, YUCORN], the hint-passing approaches of [CHAKA, HAAS, ABG, JCL, COL] and the predictive approach of [PAZDO]). This approach can work well in circumstances where re-reference by the same plan is the main factor in buffering. In Example 1.2 above, we would presumably know enough to drop pages read in by sequential scans. The DBMIN algorithm would also deal well with the references of Example 1.1 if the entire reference string were produced by a single query. However, the query plans for the simple multi-user transactions of Example 1.1 give no preference to retaining B-tree pages or data pages in buffer, since each page is referenced exactly once during the plan. In multi-user situations, query optimizer plans can overlap in complicated ways, and the query optimizer advisory algorithms do not tell us how to take such overlap into account. A more global page replacement policy must exist to make such a decision.

1.2 Contribution of the Paper

Both of the above categories of solutions take the viewpoint that, since LRU does not discriminate well between frequently and infrequently referenced pages, it is necessary to have some other agent provide hints of one kind or another. The contribution of the current paper is to derive a new self-reliant page-replacement algorithm that takes into account more of the access history for each page, to better discriminate pages that should be kept in buffer. This seems a sensible approach since the page history used by the LRU algorithm is quite limited: simply the time of last reference.

In this paper we carefully examine the idea of taking into account the history of the last two references, or more generally the last K references, $K \geq 2$. The specific algorithm developed in this paper that takes into account knowledge of the last two references to a page is named LRU-2, and the natural generalization is the LRU-K algorithm; we refer to the classical LRU algorithm within this taxonomy as LRU-1. It turns out that, for $K > 2$, the LRU-K algorithm provides somewhat improved performance over LRU-2 for stable patterns of access, but is less responsive to changes in access patterns, an important consideration for some applications.

Despite the fact that the LRU-K algorithm derives its benefits from additional information about page access frequency, LRU-K is fundamentally different from the Least Frequently Used (LFU) replacement algorithm. The crucial difference is that LRU-K has a built-in notion of "aging", considering only the last K references to a page, whereas the LFU algorithm has no means to discriminate recent versus past reference frequency of a page, and is therefore unable to cope with evolving access patterns. LRU-K is also quite different from more sophisticated LFU-based buffering algorithms that employ aging schemes based on reference counters. This category of algorithms, which includes, for example, GCLOCK and variants of LRD [EFFEHAER], depends critically on a careful choice of various workload-dependent parameters that guide the aging process. The LRU-K algorithm, on the other hand, does not require any manual tuning of this kind.

The LRU-K algorithm has the following salient properties:

- It discriminates well between page sets with different levels of reference frequency (e.g., index pages vs. data pages). Thus, it approaches the effect of assigning page sets to different buffer pools of specifically tuned sizes. In addition, it adapts itself to evolving access patterns.
- It detects locality of reference within query executions, across multiple queries in the same transaction, and also locality across multiple transactions in a multi-user environment.
- It is self-reliant in that it does not need any external hints.
- It is fairly simple and incurs little bookkeeping overhead.

The remainder of this paper has the following outline. In Section 2, we present the basic concepts of the LRU-K approach to disk page buffering. In Section 3 we give informal arguments that the LRU-K algorithm is optimal in a certain well defined sense, given knowledge of the most recent K references to each page. In Section 4, we present simulation performance results for LRU-2 and LRU-K in comparison with LRU-1. Section 5 has concluding remarks.

2. Concepts of LRU-K Buffering

In the current paper we take a statistical view of page reference behavior, based on a number of the assumptions from the *Independent Reference Model* for paging, in Section 6.6 of [COFFDENNN]. We start with an intuitive formulation; the more complete mathematical development is covered in [OOW]. Assume we are given a set $N = \{1, 2, \ldots, n\}$ of disk pages, denoted by positive integers, and that the database system under study makes a succession of references to these pages specified by the *reference string:* $r_1, r_2, \ldots, r_t, \ldots$, where $r_t = p$ $(p \in N)$ means that term numbered t in the references string refers to disk page p. Note that in the original model of [COFFDENNN], the reference string represented the page references by a single user process, so the assumption that the string reflects all references by the system is a departure. In the following discussion, unless otherwise noted, we will measure all time intervals in terms of counts of successive page accesses in the reference string, which is why the generic term subscript is denoted by 't'. At any given instant t, we assume that each disk page p has a well defined probability, b_p, to be the next page referenced by the system: $Pr(r_{t+1} = p) = b_p$, for all $p \in N$. This implies that the reference string is probabilistic, a sequence of random variables. Changing access patterns may alter these page reference probabilities, but we assume that the probabilities b_p have relatively long periods of stable values, and start with the assumption that the probabilities are unchanging for the length of the reference string; thus we assume that b_p is independent of t.

Clearly, each disk page p has an expected reference interarrival time, I_p, the time between successive occurrences of p in the reference string, and we have: $I_p = b_p^{-1}$. We intend to have our database system use an approach based on Bayesian statistics to estimate these interarrival times from observed references. The system then attempts to keep in memory buffers only those pages that seem to have an interarrival time to justify their residence, i.e. the pages with shortest access interarrival times, or equivalently greatest probability of reference. This is a statistical approximation to the A_0 algorithm of [COFFDENNN], which was shown to be optimal. The LRU-1 (classical LRU) algorithm can be thought of as taking such a statistical approach, keeping in memory only those pages that seem to have the shortest interarrival time; given the limitations of LRU-1 information on each page, the best estimate for interarrival time is the time interval to prior reference, and pages with the shortest such intervals are the ones kept in buffer.

Definition 2.1. Backward K-distance $b_t(p,K)$. Given a reference string known up to time t, $r_1, r_2, \ldots, r_t$, the backward K-distance $b_t(p,K)$ is the distance backward to the K^{th} most recent reference to the page p:

$b_t(p,K) = x$, if r_{t-x} has the value p and there have been exactly K-1 other values i with $t - x < i \leq t$, where $r_i = p$,

$= \infty$, if p does not appear at least K times in $r_1, r_2, \ldots, r_t$

Definition 2.2. LRU-K Algorithm. The LRU-K Algorithm specifies a page replacement policy when a buffer slot is needed for a new page being read in from disk: the page p to be dropped (i.e., selected as a replacement victim) is the one whose Backward K-distance, $b_t(p,K)$, is the maximum of all pages in buffer. The only time the choice is ambiguous is when more than one page has $b_t(p,K) = \infty$. In this case, a subsidiary policy may be used to select a replacement victim among the pages with infinite Backward K-distance; for example, classical LRU could be employed as a subsidiary policy. Note that LRU-1 corresponds to the classical LRU algorithm.□

The LRU-2 algorithm significantly improves on LRU-1 because by taking into account the last two references to a page, we are able for the first time to estimate I_p by measurement of an actual interarrival between references, rather than estimating simply by a lower bound of the time back to the most recent reference. We are using more information and our estimates are immensely improved as a result, especially as regards pages that have long reference interarrival time and should therefore be dropped from buffer quickly. Note that we make no general assumptions about the probabilistic distribution of I_p. In the full paper [OOW], we assume an exponential distribution for I_p to demonstrate optimality of the LRU-K algorithm. As already mentioned, we model b_p and therefore I_p as having the potential for occasional changes over time, only assuming that changes are infrequent enough that a statistical approach to future page access based on past history is usually valid. These assumptions seem justified for most situations that arise in database use.

2.1. Realistic Assumptions In DB Buffering

The general LRU-K algorithm has two features, peculiar to the cases where $K \geq 2$, that require careful consideration to ensure proper behavior in realistic situations. The first, known as *Early Page Replacement*, arises in situations where a page recently read into memory buffer does not merit retention in buffer by standard LRU-K criteria, for example because the page has a $b_t(p,K)$ value of infinity. We clearly want to drop this page from buffer relatively quickly, to save memory resources for more deserving disk pages. However we need to allow for the fact that a page that is not generally popular may still experience a burst of correlated references shortly after being referenced for the first time. We deal with this concern in Section 2.1.1. A second feature that we need to deal with in cases where $K \geq 2$, is the fact that there is a need to retain a history of references for pages that are not currently present in buffer. This is a departure from current page replacement algorithms, and will be referred to as the *Page Reference Retained Information Problem*, covered below in Section 2.1.2. A pseudo-code outline of the LRU-K buffering algorithm which deals with the concerns mentioned above is given in Section 2.1.3.

2.1.1. Early Page Replacement and the Problem of Correlated References

To avoid the wasteful use of memory buffers seen in Examples 1.1 and 1.2, LRU-K makes a decision whether to drop a page p from residence after a short time-out period from its most recent reference. A canonical period might be 5 seconds. To demonstrate the need for such a time-out, we ask the following question: What do we mean by the last *two* references to a page? We list below four ways that a pair of references might take place to the same disk page; the first three of these are called correlated reference pairs, and are likely to take place in a short span of time.

(1) **Intra-Transaction**. A transaction accesses a page, then accesses the same page again before committing. This is likely to happen with certain update transactions, first reading a row and later updating a value in the row.

(2) **Transaction-Retry**. A transaction accesses a page, then aborts and is retried, and the retried transaction accesses the page again for the same purpose.

(3) **Intra-Process**. A transaction references a page, then commits, and the next transaction by the same process accesses the page again. This pattern of access commonly arises in batch update applications, which update 10 records in sequence, commit, then start again by referencing the next record on the same page.

(4) **Inter-Process**. A transaction references a page, then a (frequently different) process references the same page for independent reasons. (At least while we do not have a great deal of communication between processes where information is passed from one process to the other in database records we can assume references by different processes are independent.)

Recall that our purpose in buffering disk pages in memory is to retain pages with relative long-term popularity to save disk I/O. An example of such long-term popularity is given in Example 1.1, where the 100 B-tree leaf pages are frequently referenced by concurrently acting transactions. The point about correlated reference-pair types (1) through (3) above is that if we take these reference pairs into account in estimating interarrival time I_p, we will often arrive at invalid conclusions. For example, reference-pair type (1) may be a common pattern of access, so if we drop a page from buffer right away after the first reference of type (1) because we have not seen it before, we will probably have to read it in again for the second reference. On the other hand, after the second reference, with the transaction committed, if we say that this page has a short interarrival time and keep it around for a hundred seconds or so, this is likely to be a mistake; the two correlated references are insufficient reason to conclude that independent references will occur. There are several obvious ways to address the problems of correlated references, the most basic of which is this: the system should *not* drop a page immediately after its first reference, but should keep the page around for a short period until the likelihood of a dependent follow-up reference is minimal; then the page can be dropped. At the same time, interarrival time should be calculated based on non-correlated access pairs, where each successive access by the same process within a time-out period is assumed to be correlated: the relationship is transitive. We refer to this approach, which associates correlated references, as the *Time-Out Correlation* method; and we refer to the time-out period as the *Correlated Reference Period* The idea is not new; in [ROBDEV] an equivalent proposal is made in Section 2.1, under the heading: Factoring out Locality.

The implication of a Correlated Reference Period of this kind on the mathematical formulation is simply this. The reference string, $r_1, r_2, \ldots, r_t$, is redefined each time the most recent reference r_t passes through the time-out period, in order to collapse any sequence of correlated references to a time interval of zero. If a reference to a page p is made several times during a Correlated Reference Period, we do not want to penalize or credit the page for that. Basically, we estimate the interarrival time I_p by the time interval from the end of one Correlated Reference Period to the beginning of the next. It is clearly possible to distinguish processes making page references; for simplicity, however, we will assume in what follows that references are not distinguished by process, so any reference pairs within the Correlated Reference Period are considered correlated.

Another alternative is to vary the Time-Out Correlation approach based on more knowledge of system events. For example, we could say that the time-out period ends after the transaction that accessed it and the following transaction from the same process commit successfully (to rule out cases (1) and (3) above), or else after a retry of the first transaction has been abandoned (to rule out case (2)); how-

ever there might be other correlated reference pair scenarios not covered by these three cases. Another idea is to allow the DBA to override a default Correlated Reference Period by setting a parameter for a particular table being processed.

2.1.2. The Page Reference Retained Information Problem

We claim that there is a need in the LRU-K algorithm, where $K \geq 2$, to retain in memory a history of references for pages that are not themselves present in buffer, a departure from most buffer replacement algorithms of the past. To see why this is so, consider the following scenario in the LRU-2 algorithm. Each time a page p is referenced, it is made buffer resident (it might already be buffer resident), and we have a history of at least one reference. If the prior access to page p was so long ago that we have no record of it, then after the Correlated Reference Period we say that our estimate of $b_t(p,2)$ is infinity, and make the containing buffer slot available on demand. However, although we may drop p from memory, we need to keep history information about the page around for awhile; otherwise we might reference the page p again relatively quickly and once again have no record of prior reference, drop it again, reference it again, etc. Though the page is frequently referenced, we would have no history about it to recognize this fact. For this reason, we assume that the system will maintain history information about any page for some period after its most recent access. We refer to this period as the *Retained Information Period.*

If a disk page p that has never been referenced before suddenly becomes popular enough to be kept in buffer, we should recognize this fact as long as two references to the page are no more than the Retained Information Period apart. Though we drop the page after the first reference, we keep information around in memory to recognize when a second reference gives a value of $b_t(p,2)$ that passes our LRU-2 criterion for retention in buffer. The page history information kept in a memory resident data structure is designated by HIST(p), and contains the last two reference string subscripts i and j, where $r_i = r_j = p$, or just the last reference if only one is known. The assumption of a memory resident information structure may require a bit of justification. For example, why not keep the information about the most recent references in the header of the page itself? Clearly any time the information is needed, the page will be buffer resident. The answer is that such a solution would require that the page always be written back to disk when dropped from buffer, because of updates to HIST(p); in applications with a large number of read-only accesses to infrequently referenced pages, which could otherwise simply be dropped from buffer without disk writes, this would add a large amount of overhead I/O.

To size the Retained Information Period, we suggest using the Five Minute Rule of [GRAYPUT] as a guideline. The cost/benefit tradeoff for keeping a 4 Kbyte page p in memory buffers is an interarrival time I_p of about 100 seconds. Returning to discussion of LRU-2, a little thought suggests that the Retained Information Period should be about twice this period, since we are measuring how far back we need to go to see *two* references before we drop the page. So a canonical value for the Retained Information Period could be about 200 seconds. We believe that this is a reasonable rule of thumb for most database applications. High-performance applications may, however, choose to increase the buffer pool beyond the economically oriented size that would follow from the Five Minute Rule. In such applications, the Retained Information Period should be set higher accordingly. To determine a reasonable value, consider the maximum Backward K-distance of all pages that we want to ensure to be memory-resident. This value is an upper bound for the Retained Information Period, because no conceivable string of new references to a page after this period will enable the page to pass the criterion for retention in buffer.

2.1.3. Schematic Outline of the LRU-K Buffering Algorithm

The LRU-K algorithm of Figure 2.1 is based on the following data structures:

- HIST(p) denotes the history control block of page p; it contains the times of the K most recent references to page p, discounting correlated references: HIST(p,1) denotes the time of last reference, HIST(p,2) the time of the second to the last reference, etc.
- LAST(p) denotes the time of the most recent reference to page p, regardless of whether this is a correlated reference or not.

These two data structures are maintained for all pages with a Backward K-distance that is smaller than the Retained Information Period. An asynchronous demon process should purge history control blocks that are no longer justified under the retained information criterion.

Based on these data structures, a conceptual outline of the LRU-K algorithm in pseudo-code form is given in Figure 2.1. Note that this outline disregards I/O latency; a real implementation would need more asynchronous units of work. Also, to simplify the presentation, the outline disregards additional data structures that are needed to speed up search loops; for example, finding the page with the maximum Backward K-distance would actually be based on a search tree. Despite the omission of such details, it is obvious that the LRU-K algorithm is fairly simple and incurs little bookkeeping overhead.

The algorithm works as follows. When a page already in the buffer is referenced, we need only update HIST and LAST for the page. In fact, if the page has been referenced last within the Correlated Reference Period, we need only update its LAST to extend the ongoing correlated reference period; otherwise a significant gap in references has occurred and we need to close out the old correlated reference period and start a new one. To close out the old period, we calculate its length in time, LAST(p) - HIST(p,1), a period of correlated-reference time that we need to shrink to a point. This contraction pulls the earlier HIST(p,i) values (i = 2,...,K) forward in time by this amount. In the same

loop, they are being pushed down a slot in the HIST array to accommodate the new HIST value involved in starting a new correlated reference period. Finally LAST is updated to the current time.

When a page not in the buffer is referenced, a replacement victim must be found to free up a buffer slot. Buffer pages currently within a correlated-reference period are ineligible for replacement, and amoung the rest, we select the one with the maximum backward K-distance, $b_t(q,K)$, or in the current notation, the minimum HIST(q,K). This victim is dropped from the buffer, possibly requiring write-back. Then a HIST block is initialized or updated for the newly referenced page.

```
Procedure to be invoked upon reference to page p at time t:

if p is already in the buffer
then   /* update history information of p */
  if t - LAST(p) > Correlated_Reference_Period
  then   /* a new, uncorrelated reference */
    correl_period_of_refd_page := LAST(p) - HIST(p,1)
    for i := 2 to K do
      HIST(p,i) := HIST(p,i-1)  +
                        correl_period_of_refd_page
    od
    HIST (p,1) := t
    LAST(p) := t
  else   /* a correlated reference */
    LAST(p) := t
  fi
else   /* select replacement victim */
  min := t
  for all pages q in the buffer do
    if t - LAST(q) > Correlated_Reference_Period
                              /* if eligible for replacement */
      and HIST(q,K) < min
                    /* and max Backward K-distance so far */
    then
      victim := q
      min := HIST(q,K)
    fi
  od
  if victim is dirty then
              write victim back into the database fi
  /* now fetch the referenced page */
  fetch p into the buffer frame previously held by victim
  if HIST(p) does not exist
  then   /* initialize history control block */
    allocate HIST(p)
    for i := 2 to K do HIST(p,i) := 0 od
  else
    for i := 2 to K do HIST(p,i) := HIST(p,i-1) od
  fi
  HIST(p,1) := t
  LAST(p) := t
fi
```

Figure 2.1. Pseudo-code Outline of the LRU-K Buffering Algorithm, Explained in Section 2.1.3

3. Optimality of the LRU-K Algorithm Under The Independent Reference Model

In this section, we give informal arguments that the LRU-2 algorithm provides essentially optimal buffering behavior based on the information given; this result generalizes easily to LRU-K. A mathematical analysis of the behavior of the LRU-K algorithm, including a proof of optimality, is given in [OOW]. In the following, we will assume for simplicity that the Correlated Reference Period is zero, and that this causes no ill effects; essentially we assume that correlated references have been factored out.

As before, we take as a starting point for our discussion the *Independent Reference Model* for paging, presented in Section 6.6 of [COFFDENN]. We take our notation from this reference and make a few departures from the model presented there. We begin with a set $N = \{1, 2, \ldots, n\}$ of disk pages and a set $M = \{1, 2, \ldots, m\}$ of memory buffers, $1 \le m \le n$. A system's paging behavior over time is described by its (page) reference string: $r_1, r_2, \ldots, r_t, \ldots$, where $r_t = p$ means that disk page p, $p \in N$, is referenced by the system at time t. According to the independent reference assumption, the reference string is a sequence of independent random variables with the common stationary distribution $\{\beta_1, \beta_2, \ldots, \beta_n\}$, one probability for each of the n disk pages, where $Pr(r_t = p) = \beta_p$, for all $p \in N$ and all subscripts t. Let the random variable $d_t(p)$ denote the time interval forward to the next occurrence of p in the reference string after r_t; from the assumptions above, $d_t(p)$ has the stationary geometric distribution:

$$(3.1) \qquad Pr(d_t(p) = k) = \beta_p (1 - \beta_p)^{k-1}, k = 1, 2, \ldots$$

with mean value $I_p = 1/\beta_p$. Note that because of the stationary assumption, $Pr(d_t(p) = k)$ is independent of t.

Definition 3.1. The A_0 Buffering Algorithm. Let A_0 denote the buffering algorithm that replaces the buffered page p in memory whose expected value I_p is a maximum, i.e., the page for which β_p is smallest. ⌘

Theorem 3.2. [COFFDENN] [ADU]
Algorithm A_0 is optimal under the independent reference assumption. ⌘

Now consider a reference string $\omega = r_1, r_2, \ldots, r_t, \ldots$, with some reference probability vector $\boldsymbol{\beta} = \{\beta_1, \beta_2, \ldots, \beta_n\}$ for the n disk pages of N. To reflect the normal state of ignorance concerning reference probabilities with which a buffering algorithm starts, we cannot assume foreknowledge of the β_i. In this situation, the best we can do is to statistically estimate β_i for each page i, based on a history of references to the page. It turns out that the analysis to derive a statistical estimate of β_i allows us to derive certain ordinal properties of these quantities. In particular, given any reference probability vector $\boldsymbol{\beta}$ with at least two distinct values, we are able to conclude that for any two disk pages

x and y, if $b_t(x,K) < b_t(y,K)$, then the page x has a higher estimate for probability of reference.

This result does not allow us to estimate the absolute values of the various β_i, but it is sufficient to order the β_i values of different pages and to determine the page with the lowest β_i value. Among all pages currently held in buffer, the page x with the highest backward K-distance has the lowest estimated reference probability. Note that this is the best possible statistical estimation of the ordinal properties of the β_i values, given only the knowledge of the last K references to a page.

We know from Theorem 3.2 that it is optimal to replace the page with the lowest reference probability whenever we need to drop a page in order to free up a buffer frame for a new page; this is the A_0 algorithm. It seems reasonable that the best possible approximation of A_0 would have us drop the page x with the lowest estimated reference probability, which is the page with the highest backward K-distance according to the above argument; this is the LRU-K algorithm. In fact, the main result of [OOW] is very close to this reasonable-sounding statement:

Theorem 3.3. [OOW]
Under the independent page reference assumption, and given knowledge of the last K references to the pages in buffer, the expected cost resulting from the LRU-K algorithm acting with m memory buffers is less than that resulting from any other algorithm acting with m-1 buffers. ⌘

Thus LRU-K acts optimally in all but (perhaps) one of its m buffer slots, an insignificant coast increment for large m. Note that this result on LRU-K applies equally to LRU-1. In particular, the LRU algorithm is seen to act optimally (under the independent page reference assumption), given the limited knowledge it has of the most recent reference time.

4. Performance Characteristics.

A prototype implementation of the LRU-2 algorithm was funded by the Amdahl Corporation to investigate optimal alternatives for efficient buffer behavior in the Huron database product. Minor alterations in the prototype permitted us to simulate LRU-K behavior, $K \geq 1$, in several situations of interest. We investigated three types of workload situations:

- a synthetic workload with references to two pools of pages that have different reference frequencies, modeling Example 1.1,
- a synthetic workload with random references to a set of pages with a Zipfian distribution of reference frequencies, and
- a real-life OLTP workload sample with random, sequential, and navigational references to a CODASYL database.

These three experiments are discussed in the following three subsections.

4.1 Two Pool Experiment

We considered two pools of disk pages, Pool 1 with N_1 pages and Pool 2 with N_2 pages, with $N_1 < N_2$. In this *two pool experiment*, alternating references are made to Pool 1 and Pool 2; then a page from that pool is randomly chosen to be the sequence element. Thus each page of Pool 1 has a probability of reference $b_1 = 1/(2N_1)$ of occurring as any element of the reference string w, and each page of Pool 2 has probability $b_2 = 1/(2N_2)$. This experiment is meant to model the alternating references to index and record pages of Example 1.1: I1, R1, I2, R2, I3, R3, ... We wish to demonstrate how LRU-K algorithms with varying K discriminate between pages of the two pools, and how well they perform in retaining the more frequently referenced pool pages (the *hotter* pool pages) in buffer. The buffer hit ratios for the various algorithms in identical circumstances give us a good measure of the effectiveness of the LRU-K algorithm, for varying K. The optimal algorithm A_0 which automatically keeps the maximum possible set of pool 1 pages buffer resident was also measured.

The buffer hit ratio for each algorithm was evaluated by first allowing the algorithm to reach a quasi-stable state, dropping the initial set of $10 \cdot N_1$ references, and then measuring the next $T = 30 \cdot N_1$ references. If the number of such references finding the requested page in buffer is given by h, then the cache hit ratio C is given by:

$$C = h / T$$

In addition to measuring cache hit ratios, the two algorithms LRU-1 and LRU-2 were also compared in terms of their cost/performance ratios, as follows. For a given N_1, N_2 and buffer size B(2), if LRU-2 achieves a cache hit ratio C(2), we expect that LRU-1 will achieve a smaller cache hit ratio. But by increasing the number of buffer pages available, LRU-1 will eventually achieve an equivalent cache hit ratio, and we say that this happens when the number of buffer pages equals B(1). Then the ratio B(1)/B(2), of buffer sizes which give the same effective hit ratio, is a measure of comparable buffering effectiveness of the two algorithms. We expect that $B(1)/B(2) > 1.0$, and a value of 2.0, for example, indicates that while LRU-2 achieves a certain cache hit ratio with B(2) buffer pages, LRU-1 must use twice as many buffer pages to achieve the same hit ratio.

The results of this simulation study are shown in Table 4.1, where $N_1 = 100$ and $N_2 = 10{,}000$.

B	LRU-1	LRU-2	LRU-3	A_0	B(1)/B(2)
60	0.14	0.291	0.300	0.300	2.3
80	0.18	0.382	0.400	0.400	2.6
100	0.22	0.459	0.495	0.500	3.0
120	0.26	0.496	0.501	0.501	3.3
140	0.29	0.502	0.502	0.502	3.2
160	0.32	0.503	0.503	0.503	2.8
180	0.34	0.504	0.504	0.504	2.5
200	0.37	0.505	0.505	0.505	2.3
250	0.42	0.508	0.508	0.508	2.2
300	0.45	0.510	0.510	0.510	2.0
350	0.48	0.513	0.513	0.513	1.9
400	0.49	0.515	0.515	0.515	1.9
450	0.50	0.517	0.518	0.518	1.8

Table 4.1. Simulation results of the two pool experiment, with disk page pools of $N_1 = 100$ pages and $N_2 = 10{,}000$ pages. The first column shows the buffer size B. The second through fifth columns show the hit ratios of LRU-1, LRU-2, LRU-3, and A_0. The last column shows the equi-effective buffer size ratio B(1)/B(2) of LRU-1 vs. LRU-2.

Consider B(1)/B(2) on the top row of Table 4.1. The B(2) value corresponds to the B of that row, 60, where we measure LRU-2 having a cache hit ratio of 0.291; to achieve the same cache hit ratio with LRU-1 requires approximately 140 pages (therefore B(1) = 140), and so 2.3 = 140/60. LRU-2 outperforms LRU-1 by more than a factor of 2 with respect to this cost/performance metric. We also note from this experiment that the results of LRU-3 are even closer to those of the optimum policy A_0, compared to the results of LRU-2. In fact, it is possible to prove, with stable page access patterns, that LRU-K approaches A_0 with increasing value of K. For evolving access patterns, however, LRU-3 is less responsive than LRU-2 in the sense that it needs more references to adapt itself to dynamic changes of reference frequencies. For this reason, we advocate LRU-2 as a generally efficient policy. The general LRU-K with K > 2, could be of value for special applications, but this requires further investigation.

For readers who feel that pools of 100 pages and 10,000 pages, as well as a buffer count B in the range of 100 are unrealistically small for modern applications, note that the same results hold if all page numbers, N_1, N_2 and B are multiplied by 1000. The smaller numbers were used in simulation to save effort.

4.2 Zipfian Random Access Experiment

The second experiment investigated the effectiveness of LRU-K for a single pool of pages with skewed random access. We generated references to N = 1000 pages (numbered 1 through N) with a Zipfian distribution of reference frequencies; that is, the probability for referencing a page with page number less than or equal to i is $(i / N)^{\log a / \log b}$ with constants a and b between 0 and 1 [CKS, KNUTH, p. 398]. The meaning of the constants a and b is that a fraction a of the references accesses a fraction b of the N pages (and the same relationship holds recursively within the fraction b of hotter pages and the fraction 1-b of colder pages). Table 4.2 compares the buffer hit ratios for LRU-1, LRU-2, and A_0 at different buffer sizes, as well as the equi-effective buffer size ratio B(1)/B(2) of LRU-1 versus LRU-2 for a = 0.8 and b = 0.2 (i.e., 80-20 skew).

B	LRU-1	LRU-2	A_0	B(1)/B(2)
40	0.53	0.61	0.640	2.0
60	0.57	0.65	0.677	2.2
80	0.61	0.67	0.705	2.1
100	0.63	0.68	0.727	1.6
120	0.64	0.71	0.745	1.5
140	0.67	0.72	0.761	1.4
160	0.70	0.74	0.776	1.5
180	0.71	0.73	0.788	1.2
200	0.72	0.76	0.825	1.3
300	0.78	0.80	0.846	1.1
500	0.87	0.87	0.908	1.0

Table 4.2. Simulation results on buffer cache hit ratios for random access with Zipfian 80-20 distribution to a disk page pool of N=1000 pages.

As in the two pool experiment of Section 4.1, LRU-2 achieved significant improvements in terms of the hit ratio at a fixed buffer size and also in terms of the cost/performance ratio. Compared to the results of Section 4.1, the gains of LRU-2 are a littler lower, because the skew of this Zipfian random access experiment is actually milder than the skew of the two pool experiment. (The two pool workload of Section 4.1 roughly corresponds to a = 0.5 and b = 0.01; however, within the b and 1-b fractions of pages, the references are uniformly distributed.)

4.3 OLTP Trace Experiment

The third experiment was based on a one-hour page reference trace of the production OLTP system of a large bank. This trace contained approximately 470,000 page references to a CODASYL database with a total size of 20 Gigabytes. The trace was fed into our simulation model, and we compared the performance of LRU-2, classical LRU-1, and also LFU. The results of this experiment, hit ratios for different buffer sizes B and the equi-effective buffer size ratio B(1)/B(2) of LRU-1 versus LRU-2, are shown in Table 4.3.

LRU-2 was superior to both LRU and LFU throughout the spectrum of buffer sizes. At small buffer sizes ($\leq$ 600), LRU-2 improved the buffer hit ratio by more than a factor of 2, compared to LRU-1. Furthermore, the B(1)/B(2) ratios in this range of buffer sizes show that LRU-1 would have to increase the buffer size by more than a factor of 2 to achieve the same hit ratio as LRU-2.

B	LRU-1	LRU-2	LFU	B(1)/B(2)
100	0.005	0.07	0.07	4.5
200	0.01	0.15	0.11	3.25
300	0.02	0.20	0.15	3.0
400	0.06	0.23	0.17	2.75
500	0.09	0.24	0.19	2.4
600	0.13	0.25	0.20	2.16
800	0.18	0.28	0.23	1.9
1000	0.22	0.29	0.25	1.6
1200	0.24	0.31	0.27	1.66
1400	0.26	0.33	0.30	1.5
1600	0.29	0.34	0.31	1.5
2000	0.31	0.36	0.33	1.3
3000	0.38	0.40	0.39	1.1
5000	0.46	0.47	0.44	1.05

Table 4.3. Simulation results on buffer cache hit ratios using an OLTP trace.

The performance of LFU was surprisingly good. The LFU policy to keep the pages with the highest reference frequency is indeed the right criterion for stable access patterns. However, the inherent drawback of LFU is that it never "forgets" any previous references when it compares the priorities of pages; so it does not adapt itself to evolving access patterns. For this reason, LFU performed still significantly worse than the LRU-2 algorithm, which dynamically tracks the *recent* reference frequencies of pages. Note, however, that the OLTP workload in this experiment exhibited fairly stable access patterns. In applications with dynamically moving hot spots, the LRU-2 algorithm would outperform LFU even more significantly.

At large buffer sizes ($\geq$ 3000), the differences in the hit ratios of the three policies became insignificant. So one may wonder if the superiority of LRU-2 at small buffer sizes is indeed relevant. The answer to this question is in the characteristics of the OLTP trace (which is probably quite typical for a large class of application workloads). The trace exhibits an extremely high access skew for the hottest pages: for example, 40% of the references access only 3% of the database pages that were accessed in the trace. For higher fractions of the references, this access skew flattens out: for example, 90% of the references access 65% of the pages, which would no longer be considered as heavily skewed. An analysis of the trace showed that only about 1400 pages satisfy the criterion of the Five Minute Rule to be kept in memory (i.e., are re-referenced within 100 seconds, see Section 2.1.2). Thus, a buffer size of 1400 pages is actually the economically optimal configuration. There is no point in increasing the buffer size to keep additional pages once locality flattens out. The LRU-2 algorithm keeps this pool of 1400 hot pages memory resident, at a memory cost of only two thirds of the cost of the classical LRU algorithm (i.e., B(1)/B(2) = 1.5 for B=1400).

5. Concluding Remarks

In this paper we have introduced a new database buffering algorithm named LRU-K. Our simulation results provide evidence that the LRU-K algorithm has significant cost/performance advantages over conventional algorithms like LRU, since LRU-K can discriminate better between frequently referenced and infrequently referenced pages. Unlike the approach of manually tuning the assignment of page pools to multiple buffer pools, our algorithm is self-reliant in that it does not depend on any external hints. Unlike the approaches that aim to derive hints to the buffer manager automatically from the analysis of query execution plans, our algorithm considers also inter-transaction locality in multi-user systems. Finally, unlike LFU and its variants, our algorithm copes well with evolving access patterns such as moving hot spots.

One of the new concepts of our approach is that page history information is kept past page residence. But clearly this is the only way we can guarantee that a page referenced with metronome-like regularity at intervals just above its residence period will ever be noticed as referenced twice. It is an open issue how much space we should set aside for history control blocks of non-resident pages. While estimates for an upper bound can be derived from workload properties and the specified Retained Information Period, a better approach would be to turn buffer frames into history control blocks dynamically, and vice versa.

The development of the LRU-K algorithm was mostly motivated by OLTP applications, decision-support applications on large relational databases, and especially combinations of these two workload categories. We believe, however, that the potential leverage of our algorithm may be even higher for non-conventional engineering and scientific databases. The reason is that buffer management for such applications is inherently harder because of the higher diversity of access patterns. The page pool tuning approach outlined in Section 1 is clearly infeasible for this purpose. The approaches that derive buffer manager hints from the analysis of query execution plans are questionable, too, for the following reason. Non-conventional database applications of the mentioned kind will probably make heavy use of user-defined functions, as supported by object-oriented and extensible database systems. Unlike relational queries, the access patterns of these user-defined functions cannot be pre-analyzed if the functions are coded in a general-purpose programming language (typically, in C++). Thus, advanced applications of post-relational database systems call for a truly self-reliant buffer management algorithm. The LRU-K algorithm is such a self-tuning and adaptive buffering algorithm, even in the presence of evolving access patterns. We believe that LRU-K is a good candidate to meet the challenges of next-generation buffer management.

References

[ABG] Rafael Alonso, Daniel Barbara, Hector Garcia-Molina, Data Caching Issues in an Information Retrieval System, ACM Transactions on Database Systems, v. 15, no. 3, pp. 359-384, September 1990.

[ADU] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman: Principles of Optimal Page Replacement. J. ACM, v. 18, no. 1, pp. 80-93, 1971.

[CHAKA] Ellis E. Chang, Randy H. Katz, Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS, Proceedings of the 1989 ACM SIGMOD Conference, pp. 348-357.

[CHOUDEW] Hong-Tai Chou and David J. DeWitt: An Evaluation of Buffer Management Strategies for Relational Database Systems. Proceedings of the Eleventh International Conference on Very Large Databases, pp. 127-141, August 1985.

[CKS] George Copeland, Tom Keller, and Marc Smith: Database Buffer and Disk Configuring and the Battle of the Bottlenecks. Proceedings of the Fourth International Workshop on High Performance Transaction Systems, September 1991.

[COL] C.Y. Chan, B.C. Ooi, H. Lu, Extensible Buffer Management of Indexes, Proceedings of the Eighteenth International Conference on Very Large Data Bases, pp. 444-454, August 1992.

[COFFDENN] Edward G. Coffman, Jr. and Peter J. Denning: Operating Systems Theory. Prentice-Hall, 1973.

[DANTOWS] Asit Dan and Don Towsley: An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. Proceedings of the 1990 ACM Sigmetrics Conference, v. 18, No. 1, pp. 143-149.

[DENNING] P. J. Denning: The Working Set Model for Program Behavior. Communications of the ACM, v. 11, no. 5, pp. 323-333, 1968.

[EFFEHAER] Wolfgang Effelsberg and Theo Haerder: Principles of Database Buffer Management. ACM Transactions on Database Systems, v. 9, no. 4, pp. 560-595, December 1984.

[[FNS] Christos Faloutsos, Raymond Ng, and Timos Sellis, Predictive Load Control for Flexible Buffer Allocation, Proceedings of the Seventeenth International Conference on Very Large Data Bases, pp. 265-274, September 1991.

[GRAYPUT] Jim Gray and Franco Putzolu: The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. Proceedings of the 1987 ACM SIGMOD Conference, pp. 395-398.

[HAAS] Laura M. Haas et al., Starburst Mid-Flight: As the Dust Clears, IEEE Transactions on Knowledge and Data Engineering, v. 2, no. 1, pp. 143-160, March 1990.

[JCL] R. Jauhari, M. Carey, M. Livny, Priority-Hints: An Algorithm for Priority-Based Buffer Management, Proceedings of the Sixteenth International Conference on Very Large Data Bases, August 1990.

[KNUTH] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[NFS] Raymond Ng, Christos Faloutsos, T. Sellis, Flexible Buffer Allocation Based on Marginal Gains, Proceedings of the 1991 ACM SIGMOD Conference, pp. 387-396.

[OOW] Elizabeth O'Neil, Patrick O'Neil, and Gerhard Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering, Tech. Report 92-4, Department of Mathematics and Computer Science, University of Massachusetts at Boston, December 1, 1992.

[PAZDO] Mark Palmer, Stanley B. Zdonik, Fido: A Cache That Learns to Fetch, Proceedings of the Seventeenth International Conference on Very Large Databases, pp. 255-264, September 1991.

[REITER] Allen Reiter: A Study of Buffer Management Policies for Data Management Systems. Tech. Summary Rep. No. 1619, Mathematics Research Center, Univ. of Wisconsin, Madison, March 1976.

[ROBDEV] John T. Robinson and Murtha V. Devarakonda: Data Cache Management Using Frequency-Based Replacement. Proceedings of the 1990 ACM Sigmetrics Conference, v. 18, No. 1, pp. 134-142.

[SACSCH] Giovanni Mario Sacco and Mario Schkolnick: Buffer Management in Relational Database Systems, ACM Transactions on Database Systems, v. 11, no. 4, pp. 473-498, December 1986.

[SHASHA] Dennis E. Shasha, Database Tuning: A Principled Approach, Prentice Hall, 1992.

[STON] Michael Stonebraker: Operating System Support for Database Management. Communications of the ACM, v. 24, no. 7, pp. 412-418, July 1981.

[TENGGUM] J.Z. Teng, R.A. Gumaer, Managing IBM Database 2 Buffers to Maximize Performance, IBM Systems Journal, v. 23, n. 2, pp. 211-218, 1984.

[TPC-A] Transaction Processing Performance Council (TPC): TPC BENCHMARK A Standard Specification. The Performance Handbook: for Database and Transaction Processing Systems, Morgan Kaufmann 1991.

[YUCORN] P.S. Yu, D.W. Cornell, Optimal Buffer Allocation in a Multi-query Environment, Proceedings of the Seventh International Conference on Data Engineering, pp. 622-631, April 1991.