CSC 540 – Fall 2013

Database Management Concepts and Systems

Project 2 - SimpleDB

Due date window: 11:45 AM on 27th November to 11:45 PM on 30th November

Note: The project deadline shall not be extended. Submissions made on the 27th of November shall be awarded with extra credit (1 course point).

1. **Improved buffer search**
   The SimpleDB buffer manager traverses the buffer pool sequentially when searching for pages, which is not an efficient technique. Implement a new traversal technique using data structures such as hash tables or special-purpose lists to improve search times. Document the improvement gained using suitable graphs.

2. **LRU-K Page replacement policy**
   SimpleDB considers four buffer replacement policies – Naïve, FIFO, LRU and Clock. Introduce a new replacement policy based on LRU, which functions as detailed below:

   An overview on buffer pool:
   The buffer manager is responsible for managing pages that hold user data. It allocates a fixed set of pages called the buffer pool. A page is set to be pinned, if some client is currently pinning it; otherwise it is unpinned. Clients unpin pages after use. When a new pin request arrives, the buffer manager assigns the requested blocks to unallocated pages. If there are no available pages, an existing unpinned page needs to be replaced.

   If there are no unpinned pages, clients need to wait. If there are multiple unpinned pages available, the buffer manager must decide which page to replace. This is determined by the page replacement policy.

   The LRU-K policy:
   The *LRU policy* drops the page from buffer that has not been accessed for the longest time. This policy considers limited page access information (time of last reference) and is unable to differentiate between frequent pages from infrequent ones. The ***LRU-K policy*** instead decides on what pages to keep in buffer, based on the access history for each page.

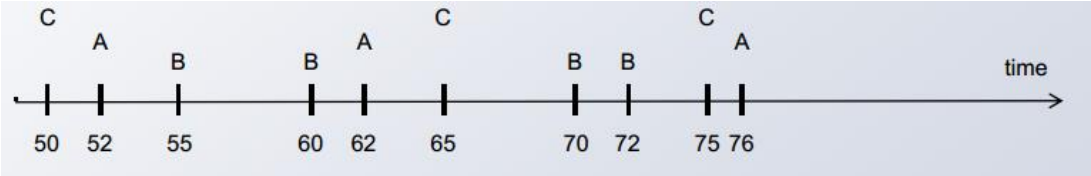
   Assume the buffer pool has *N* pages, and the database system makes successive references to these pages, specified by the reference string: $r_1, r_2, \ldots, r_t$ where $r_t = p$ ($p \in N$). Time intervals are measured in terms of counts of successive page accesses in the reference string. At any given instant t, we assume that each disk page p has a well-defined probability, to be the next page referenced by the system. The Inter-arrival time l_p is defined as the time between successive occurrences of p in the reference string. The database system uses an approach based on Bayesian statistics to estimate

these inter-arrival times from observed references. The system then attempts to keep in memory buffers only those pages that seem to have an inter-arrival time to justify their residence, i.e. the pages with shortest access inter-arrival times, or equivalently greatest probability of reference.

Example: The LRU–1 (classical LRU) algorithm keeps in memory only those pages that seem to have the shortest inter-arrival time, which is defined based on the time interval to prior reference.

Backward K–distance $b_t(p,K)$:
Given a reference string known up to time t (r_1, r_2, … , 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 <= t , where r_i = p,
= ∞, if p does not appear at least K times in r_1, r_2, … , r_t

For example, given the reference string CABBACBBCA:



We have $b_{100}(A, 3) = 48$.

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.
If $b_t(p,K)$ = ∞, then an alternative policy can be used to select a replacement victim.

Example: The LRU–2 algorithm takes into account the last two references to a page.

Code snippet for LRU-k page replacement algorithm:

```
Procedure to be invoked upon a 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 */
          correlation_period_of_referenced_page := LAST(p) -
      HIST(p,1)
          for i := 2 to K do
            HIST(p,i) := HIST(p,i-1) +
```

```
    correlation_period_of_referenced_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 /*eligible
    for replacement*/
        and HIST(q,K) < min /* maximum 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 that was 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
```

The LRU–K algorithm 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.
  HIST(p,1) denotes the last reference, HIST(p,2) the second to the last reference, …
- 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.

Implement the LRU-K algorithm for K=2 to 5.

3. **New data types**
   Modify *Class Page* to handle data types other than Integers and Strings. Specifically, create getter and setter methods for Short Integers, Booleans, Byte Arrays and Dates.

   The following data types in Java need to be supported:
   a. `short`
   b. `boolean`
   c. `byte[]`
   d. `Date`

   The following methods need to be implemented:
   a. `public synchronized short getShort(int offset)`
   b. `public synchronized void setShort(int offset, short val)`
   c. `public synchronized boolean getBoolean(int offset)`
   d. `public synchronized void setBoolean(int offset, boolean val)`
   e. `public synchronized byte[] getBytes(int offset)`
   f. `public synchronized void setBytes(int offset, byte[] val)`
   g. `public synchronized Date getDate(int offset)`
   h. `public synchronized void setDate(int offset, Date val)`

   Usage example:

```
Page p1;
…
short n = p1.getShort(100);
p1.setShort(200, n*2);
…
```

References

Paper on LRU-K: http://www.csd.uoc.gr/~hy460/pdf/p297-o_neil.pdf
SimpleDB: http://www.cs.bc.edu/~sciore/simpledb/