Roadmap

1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
3) Transactions: locking, recovery
4) Distributed DBMSs
5) Parallel DBMSs: Gamma, Alphasort
6) OODB DBMS
7) Data Analysis - data mining
8) Benchmarks
9) Vision statements
   extras (streams/sensors, graphs, multimedia, web, fractals)

Detailed Roadmap

1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
   - OS and DBMSs
   - R-trees, z-ordering
   - Buffer management: DBMIN
   - Buffer management: LRU-K
3) Transactions: locking, recovery

Outline of LRU-K

- Motivation
- Limitations of previous approaches
- Basic concepts
- Addressing realistic problems
- Algorithm

Reference

Buffering, SIGMOD 1993, pp. 297-306

Motivation

GUESS when the page will be referenced again. Problems with LRU:?
Motivation
GUESS when the page will be referenced again.
Problems with LRU:
- Makes decision based on too little info
- Cannot tell between frequent/infrequent refs on time
- System spends resources to keep useless stuff around

Example Scenario 1
- Relation CUSTOMER with 20,000 tuples
- Clustered B-tree on CUST_ID, 20b/key
- 4K pages, 4000 bytes useful space
- 100 leaf pages
- Many users
- References L1, R1, L2, R2, L3, R3, …
- Probability to ref Li is .005, to ref Ri is .00005
- LRU?

Example Scenario 2
- Relation R with 1,000,000 tuples
- A bunch of processes ref 5000 (0.5%) tuples
- A few batch processes do sequential scans
- LRU?

Related Work
- Page pool tuning (i.e., domain separation)
  - Needs constant recalibration
  - Cannot handle locality (hot spot patterns)changes
  - Hard to program
- Query execution plan analysis (hot set, DBMIN, hint-passing approaches)
  - Info from the query optimizer
  - Works well when same plan references
- DBMIN is best of the above
  - But multuser breaks it (optimizer can’t detect overlaps)

Outline of LRU-K
- Motivation
- Limitations of previous approaches
- Basic concepts
- Addressing realistic problems
- Algorithm
Basic concepts

Idea: Take into account history: last K references
(Classic LRU: K=1 (LRU-1))
(keep track of history, and try to predict)

Algorithm
• Backward K-distance b_{t}(p,K):
  #refs from t back to the Kth most recent reference to p
• b_{t}(p,K) = \infty if Kth ref doesn’t exist
• Algorithm:
  Drop page p w/ max Backward K-distance b_{t}(p,K)
• Ambiguous when infinite (use subsidiary policy, e.g., LRU)
• LRU-2 Is better that LRU-1 – Why? (I_{p})

Basic concepts (cont’d)
Parameters:
• Pages N={1,2,…,n}
• Reference string r_{1}, r_{2}, …, r_{t}, …
• r_{t}=p for page p at time t
• b_{p} = probability that r_{t+1}=p
• Time between references of p: I_{p} = 1/b_{p}

But:
• There are subtle problems:

Realistic problems
• P1: Early page replacement
  – Page b_{t}(p,K) is infinite, so drop
  – But what if it is a rare but “bursty” case?
• P2: Page reference retained information
  – For K>1- page may be gone / its information still around

P1: Early page replacement
• Should we worry about it?
P1: Early page replacement

- Should we worry about it?
- A: yes - correlated references! Examples?

Correlated References

- (1) Intra-transaction
  - E.g., read tuple/update tuple
- (2) Transaction Retry
  - Rolled back and restarted
- (3) Intra-process
  - A process references page via 2 transactions
  - E.g., update RIDs 1-10, commit, update RIDs 11-20
- (4) Inter-process
  - Two processes reference the same page independently

Outline of LRU-K

- Motivation
- Limitations of previous approaches
- Basic concepts
  - Addressing realistic problems
    - P1: Early page replacement
    - P2: reference retained information
- Algorithm

Addressing Correlation

- Problem: For example, assume (1) – read/update
  - Algorithm sees p (read)
  - Drops it (infinite b(p,K)) (wrong)
  - Sees it again (update)
  - Keeps it around (wrong again)
- Should take into account only non-correlated refs
- But how do we know?

Addressing Correlation (cont.)

- Solution: “Correlated Reference Period” by process
  - No penalty or credit for refs within CRP
  - I_C: interval from end of one CRP to begin of the next

Outline of LRU-K

- Motivation
- Limitations of previous approaches
- Basic concepts
  - Addressing realistic problems
    - P1: Early page replacement
    - P2: reference retained information
- Algorithm
P2: Reference Retained Information

- Algorithm needs to keep info for pages that may not be resident anymore, e.g.,
  - p is referenced and comes in for the first time
  - h(p,2) = infinity, p is dropped
  - p is referenced again if no info on p is retained, p may be dropped again

Reference Retained Information (cont’d)

- “Retained Information Period”
  - Period after which we drop info about page p
  - “Five minute rule” suggests RIP
- Page history information HIST(p) with <= 2 refs to p

Data Structures for LRU-K

HIST(p) – history control block of page p
  - Times of K more recent references to p
    - (correlated)
LAST(p) – time of most recent ref to page p
  - correlated references OK
- Maintained for all pages p: B_i(p,K) < RIP
- Purged asynchronously

Outline of LRU-K

- Motivation
- Limitations of previous approaches
- Basic concepts
- Addressing realistic problems
  - P1: Early page replacement
  - P2: Reference retained information

Algorithm

LRU-K Algorithm

If p is in the buffer {
  // update history of p
  if (t-LAST(p)) > CRP { // uncorrelated reference
    // close correlated period and start new
    for i=K-1 to 1
      move HIST(p,i) into slot HIST(p,i+1)
    HIST(p,1)=t
  }
  LAST(p)=t
}

LRU-K Algorithm (cont.)

else { // select replacement victim
  min=t
  for all pages q in buffer {
    if t-LAST(q)>CRP // eligible for replacement
      and HIST(q,K)<min) { // max Backward-K
      victim=q
      min=HIST(q,K)
    }
  }
  if victim dirty write back before dropping
LRU-K Algorithm (cont.)
fetch p into the victim’s buffer
if no HIST(p) exists {
    allocate HIST(p)
    for i=2 to K HIST(p,i)=0
} else {
    for i=2 to K HIST(p,i)= HIST(p,i-1)
    HIST(p,1)=t // last non-correlated reference
    LAST(p)=t // last reference
}

Two-pool Experiment
• Two disk page pools, N1=100 / N2=10,000 pages
• Models alternating index/record references
• Results
  – LRU-1 needs 2-3 times bigger BP to reach LRU-2 hit rate
  – LRU-2 really close to LRU-3 and optimal

Single-pool / Random Access
• One disk page pool, N=1000 pages
• Zipf(a,b) distribution of reference frequencies
  (fraction a of references accesses fraction b of pages)
• Results
  – LRU-2 still wins, although not by as much (milder skew)

Real OLTP Workload
• Traces from bank OLTP Xtion references
• 470,000 page references, 20GB database
• Compared to LFU as well
• Results
  – LRU-2 beats LRU-1
  – LRU-2 also beats LFU (why?)

Conclusions
• LRU not good enough
• LFU has limitations
• Other algorithms
  – too complex
  – can’t cope with change/multiple users
• LRU-K works well
• Really, LRU-2 is most beneficial
• Today: use simple algorithms, e.g., Oracle

Addendum: 2Q
(It has an excellent description of LRU-K!)
2Q - Idea

- Simpler record-keeping/tuning (CPR, RIP)
- Heart of the idea?

2Q - Idea

- Simpler record-keeping/tuning (CPR, RIP)
- Heart of the idea: Instead of keeping statistics for LRU-2, use two queues
  - Am: one LRU for ‘hot’ pages
  - A1: one FIFO for ‘not-yet-proven-hot’ pages
  - if a page from A1 is re-referenced, move to Am
- Like LRU-2: ‘scan resistant’

2Q - subtleties

- Scan: each page is referenced once; goes in and out of the A1 queue, FIFO style
- But: there is still an issue - which one?

2Q - subtleties

- How to choose the relative sizes of A1 and Am queues
- (fixed division works fine for synthetic data, but NOT for real workloads, where the hot-set size changes dynamically)
- How to resolve the issue?

2Q - final method

- Idea#1: 3 queues:
  - Am: for ‘hot’ pages
  - A1in: for pages of potentially correlated accesses
  - A1out: for pages that have been accessed once
- Idea#2:
  - A1out consists of page-ids only - not pages-slots!

2Q - final method

- Idea#1: 3 queues:
  - Am: for ‘hot’ pages
  - A1in: for pages of potentially correlated accesses
  - A1out: for pages that have been accessed once
- Idea#2:
  - A1out consists of page-ids only - not pages-slots!
2Q - drill

- is 2Q 'scan resistant'?
- r1, r2, r3, r4, ..., r1000 - how do its queues behave?

2Q - drill

- is 2Q 'scan resistant'?
- r1, r2, r3, r4, ..., r1000 - how do its queues behave?
- A:
  - Am will be empty (-> available to others)
  - Alin will have the latest pages (ri, ri-1, ...)
  - Alout will have pointers for (most of) the rest

Conclusion: it works as well as LRU-2

- with less record keeping
- faster list processing and
- fewer parameters to tune:
  - 25% of buffers to Ain;
  - A1out should have enough pointers for 50% of buffers