An Evaluation of Buffer Management Strategies for Relational Database Systems

By Hong-Tai Chou and David J. DeWitt

Junfei Geng, Dazhi Wang, Jing Zhang and Junyi Xie

CPS 216, Duke CS,
Oct 20, 2001

One-line summary:

A model of relational query behavior
(QLSM: Query Locality Set Model)

↓

A buffer management strategy: DBMIN

(QLSM predicts future reference behavior)

Outline

- Introduction and Related Work (me)
- Old Algorithms and QLSM (Dazhi)
- DBMIN (Jing)
- Evaluation, Results and Conclusions (Junyi)

Introduction and Related Work

By
Junfei Geng

(Stonebraker)

Why not use OS ?

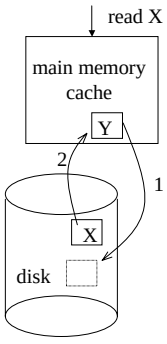
- "Operating System Support for Database Management", by Michael Stonebraker, 1981. (e.g. Unix/INGRES)
- OS fails to meet the need of DBMS: wrong service or severe performance problems.
  - Buffer pool management. (next several slides)
  - File system. (physical contiguity, overhead)
  - Scheduling, process management, and IPC. (overhead of task switching, lack of scheduling control)
  - Consistency control. (only locking for files)

(Stonebraker)

Problems with OS Buffer Pool Management

How Unix does it?

- Fixed buffer pool and all file I/O handled through this cache.
- Read/write.
- LRU replacement.
- Prefetch when UNIX detects sequential access to a file.



Review of replacement policies

Assume fixed number of frames in memory assigned to this process.

- Optimal – Belady's Algo (2)
- Random (0/1/2)
- FIFO (0)
- LRU – least recently used (0)
  To capture program locality.
- LRU approximation: Clock, Aging
- MRU – most recently used (2)

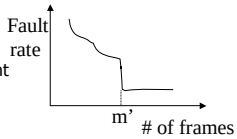
w=0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7
Now

0
1
2

3 page frames

Review of replacement policies

- Working set algorithm
  - To capture locality changes: use current memory needs to determine the number of page frames for a process.
  - Working set at t: the set of pages referenced in (t-w, t).
  - Replacement: any page in memory but not in any process's WS is a candidate.



w=0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7
t - w t
(w: window size)

(Stonebraker)

Problems with OS Buffer Pool Management

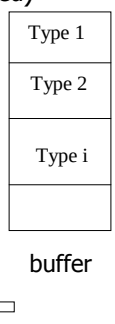
- LRU Replacement
  - Database access pattern in INGRES
    1. Seq. access to blocks which will not be revisited. (toss immediately)
    2. Seq. access to blocks which will be cyclically revisited. (MRU)
    3. Random access to blocks which will not be revisited.
    4. Random access to blocks which will possibly be revisited. (LRU)
- Prefecth rules
  DB knows what's next, but next block is not necessarily the next one in logical file order.

Previous Buffer Management Ideas ("Bad" algorithms)

- Domain separation (Reiter 76)
- "New" algorithm (Kaplan 80)
- Hot Set (Sacco and Schkolnick 82)

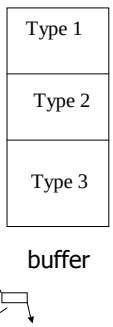
Domain Separation (page-oriented)

- Root page of the B-tree is more important.
- Pages are classified into types, each of which is separately managed in its associated domain of buffers.
- Borrow
- Inside each domain: LRU



Domain Separation

- A simple type assignment.
- 8-10% improvement than LRU.
- Limitation:
  - Static, the importance of page may vary in different queries.
  - Does not differentiate importance between types.
- Conclusion:
  - No better than global algo's, such as LRU and CLOCK. (Effelsberg and Haerder)



Old Algorithms and QLSM

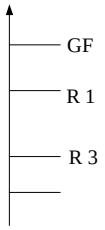
Dazhi Wang

"New" Algorithm

- Based on two observations:
  1. The priority of a page is a property of the relation to which it belongs;
  2. Each relation needs a "working set".
- Buffer pool is subdivided and allocated on a per-relation basis

"New" Algorithm

- Algorithm:
  - Each active relation is assigned a resident set which is initially empty.
  - The resident sets of relations are linked in a priority list; Unlikely reused relations are near top.
  - Ordering of relation is pre-determined, and maybe adjusted subsequently.
  - Search from top of the list
  - Within each relation: MRU.

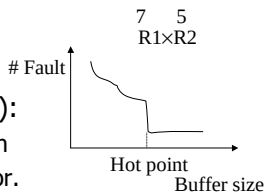


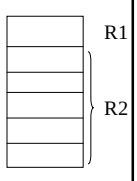
"New" Algorithm

- Pro:
  - A new approach that tracks the locality of a query through relations.
- Con:
  - The use of MRU is justifiable only in limited cases.
  - The rules for ordering relations were based on intuition.
  - Searching a list can be expensive under high memory contention.
  - Hard to extend to multi-user environment

Hot Set Algorithm

- Hot set (query behavior):
  - a set of pages over which there is a looping behavior.
- Hot point: points of discontinuity.
  e.g.: R1× R2, hot point=1+P(R2)=6
  where, R2 is inner loop.
- assign buffer for each query based on hot points; within a buffer, LRU.





Hot Set Algorithm

- Pro:
  - Provides more accurate reference pattern.
- Con:
  - Based on LRU, which is inappropriate for certain looping behavior.

QLSM: The Query Locality Set Model

- Based on the observation:
  - Relational database systems support a limited set of operations
  - Reference patterns are regular and predictable
  - The reference patterns can be decomposed into simple reference patterns
- Reference pattern classification
  - Sequential Reference
  - Random Reference
  - Hierarchical reference

Sequential Reference

- **Straight Sequential(SS):** sequential scan without repetition
  - **e.g.** selection on an unordered relation
- **Clustered Sequential(CS)**: local rescan in the course of a sequential scan
  - **e.g.** merge join

    Table1: 1 2 2 3

    Table2: 0 2 2 2 5
- **Looping Sequential(LS)**: sequential reference be repeated several times
  - **e.g.** nested loop join

Random Reference

- **Independent Random(IR):** genuinely random accesses
  - **e.g.** access data pages through a non-clustered index scan
- **Clustered Random(CR):** random accesses which demonstrate locality
  - **e.g.** join
    - Inner table: non-clustered, non-unique index
    - Outer table: clustered, non-unique keys

Hierarchical reference

- **Straight Hierarchical**(**SH**): traverse the index only once
  - **Hierarchical/Straight Sequential**(**H/SS**): traversal followed by straight sequential scan.
  - **Hierarchical/ Clustered Sequential** (**H/CS**): traversal followed by clustered sequential scan.
- **Looping Hierarchical**(**LH**): Repeatedly traverse an index.
  - **e.g.** join in which the inner relation is indexed on the join field

DBMIN Algorithm

Jing Zhang

DBMIN

A buffer management algorithm based on the QLSM.

- Per-file buffer management.
- Each file has a locality set-the set of buffers referenced for that file.
- Manage each locality set by the access pattern for that file.

Parameters

- N: total number of buffers(page frames)
- l_{ij}: max number of buffers for file instance *j* of query *i* (desired size)
- r_{ij}:number of buffers allocated for file instance *j* of query *i* (actual size)

DBMIN-Algorithm

- Initialize all buffers on global free list
- Initialize all locality sets empty with both l(maximum number of buffers allocated to a query for a particular file) and r(the number of buffers currently allocated to a query for a particular file) to 0.
- If a page is found in both the global and locality set, update usage stats.

DBMIN-Algorithm(Cont.)

- If the page is in memory, but not local set, add it to locality set(if it doesn't belong to someone else), increment r, and if r>1, evict a page according to the pattern for this pool.
- If the page isn't in memory, read it into a free buffer and proceed as in memory above.
- On file open/close, do load control:

  (Open): if $\sum_i \sum_j l_{ij} < N$, query can proceed, o/w blocks

  (Close): release buffers to free list, unblock one or more other queries

Local algorithms

- **SS**
  - Size is 1
  - Replace page as needed
- **CS**
  - Size equal (#tuples in largest cluster)/(# of tuples per page).
  - Replacement is LRU or FIFO
- **LS**
  - Size equal size of file(relation)
  - Replacement is MRU

Local algorithms(Cont.)

- **IR**
  - Replacement is whatever you want
  - Size is either 1 or threshold(b)

    b: the total number of pages referenced

    k: the number of random record accesses

    e.g.: r=k-b/b, r: residual value

    if r<=β, size=1(β is the threshold which a page is considered to have a high probability to be re-referenced).

    otherwise, size=b

Local algorithms(Cont.)

- **CR**
  - Size is the size of the # of tuples in largest cluster
  - Replacement is LRU/FIFO
- **SH, H/SS**
  - Size is 1
  - Replacement is MRU
- **H/CS**
  - Similar as CS
  - Each member in a cluster is a key-pointer rather than a data record
- **LH**
  - Size is 3-4(roughly h)
  - Replacement is LIFO

Evaluations, Results and Conclusions

By

Junyi Xie

Evaluation Methodology

- Hybrid Simulation Model
  - Trace-driven simulation
    - Describe the behavior of individual query
    - Traces recorded from a real system
  - Distribution-driven simulation
    - Events generated randomly
    - Synthesize the system workload
  - Combination of trace-driven and distribution-driven simulations
    - System Workload: described by merging all traces
    - Individual query: described by a trace string

Workloads

- Predefined Query Mixes
  - Query Mix 1 - M1
    - All six query types are equally requested
  - Query Mix 2 – M2
    - I and II are chosen half of the time
  - Query Mix 3 – M3
    - I and II have a combined probability of 75%

Query Type	CPU Demand	Disk Demand	Memory Demand
I	Low	Low	Low
II	Low	High	Low
III	High	Low	Low
IV	High	High	Low
V	High	Low	High
VI	High	High	High

Query Classification

Query Mix	Type I	Type II	Type III	Type Iv	Type V	Type VI
M1	16.67	16.67	16.67	16.67	16.67	16.67
M2	25.00	25.00	12.50	12.50	12.50	12.50
M3	37.5	37.5	6.25	6.25	6.25	6.25

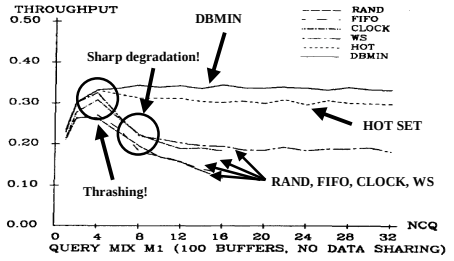
(in %)

Experiments Set One

- Configuration
  - Metrics:
    - System throughput measured by queries completed per second under certain number of concurrent queries
  - No Data Sharing
    - Every query has its own copy of data
  - No Load Control

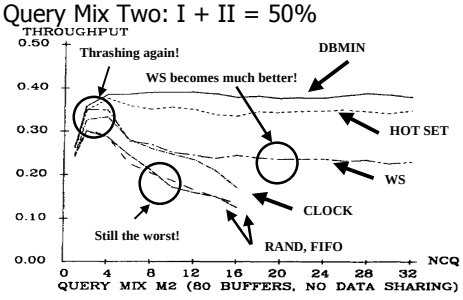
Simulation Results

Query Mix One: all queries equally distributed



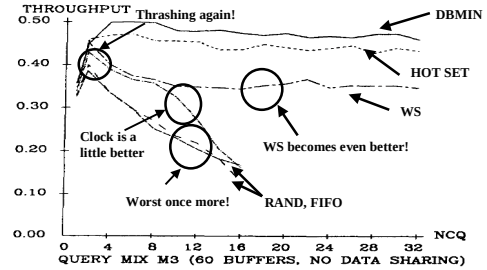
Simulation Results

Query Mix Two: I + II = 50%



Simulation Results

Query Mix Three: I + II = 75%



Simulation Results

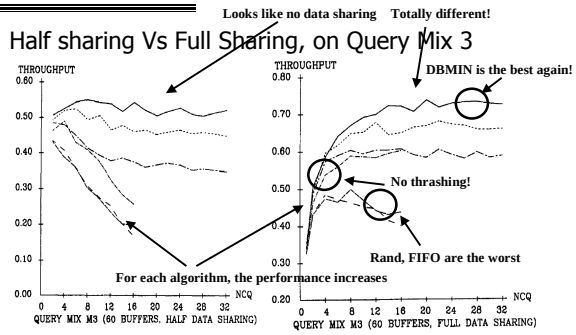
- DBMIN wins in ALL cases!
- Thrashing always occurs in FIFO, Rand and Clock.
- Performance degradation associated with FIFO, Rand and Clock.
- Rand and FIFO yield the worst performance,
- WS does not perform well in M1
  - but improved in M2, M3 where query type I, II increased.

Experiments Set Two

- Effect of Data Sharing
  - Half Sharing:
    - Every two queries share a copy of data
  - Full Sharing
    - All queries share a copy of data
  - No Load Control
- Metrics:
  - Throughput measured by queries/second

Simulation Results

Half sharing Vs Full Sharing, on Query Mix 3



Simulation Results

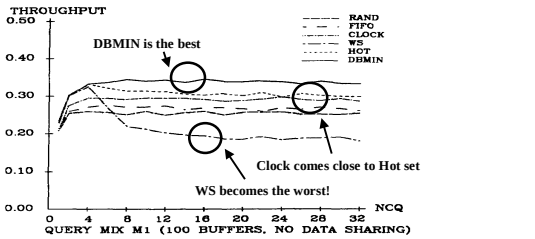
- Data sharing increases the performance for ALL algorithms.
- It eliminates thrashing, which is evident in the cases of no data sharing and half data sharing.
- DBMIN achieves the highest performance in data sharing.
- Rand and FIFO always perform the worst.

Experiments Set Three

- Effect of Load Control
  - What: Mechanism to check the usage of resources to prevent system from overloading
  - Why: To eliminate the thrashing.
  - How: "50% rule" – empirical
    - When page is kept busy about half of time, we get best performance
  - Feedback load controller
    - Estimator: measures utilization of pages
    - Optimizer: decides load adjustment to take
    - Control switch: activates/deactivates queries according to decisions from optimizer

Simulation Results

Feedback load control of Query Mix 1



Feedback Load Control

- Pros
  - Increase performance of simple algorithms
    - FIFO, Rand, Clock
- Cons
  - Runtime overhead
    - Estimator, optimizer, control switch
  - Non-predictive
    - Only respond after undesirable condition occurs

Conclusion

- DBMIN wins in ALL cases
  - Followed by Hot set, Clock, WS.
  - Rand and FIFO do not work well at all.
- Data Sharing can increase the performance, eliminate thrashing, but DBMIN and Hot set still win.
- Load control makes simple algorithms outperform WS, but there are problems with load control (overhead, non-predictive).

Weakness

- DBMIN needs to predict usage of file instance
  - But, is it predictable?
- In the case of multiple users
  - If all requirements can not satisfied, what to do?
    - Delay one(who)? – How to be fair?
    - Or Let them all suffer? – Is that fair?
- Each file instance is considered independently
  - So how to make use of the locality across file instance?
    - Across file accesses with one query

Question?