

Buffer Management

CPS 216
Advanced Database Systems

Announcements

- ❖ Reading assignment
 - Buffer management paper online
 - DB2 query rewrite paper available as hard copies
- ❖ Homework #3 was (finally) posted last night
 - Due in 14 days (Wednesday April 9)
- ❖ Recitation session this Friday (March 28)

Memory management for DBMS

- ❖ DBMS operations require main memory
 - While data resides on disk, it is manipulated in memory
 - Sometimes the more memory the better, e.g., sort
- ❖ One approach: let each operation pre-allocate some amount of “private” memory and manage it explicitly
 - Not very flexible
 - Limits sharing and reuse
- ❖ Alternative approach: use a buffer manager
 - Responsible for reading/writing data blocks from/to disk as needed
 - Higher-level code can be written without worrying about whether data is in memory or not

Buffer manager basics

- ❖ Buffer pool: a global pool of frames (main-memory blocks)
 - ☞ Some systems use separate pools for different objects (e.g., tables and indexes) and for different operations (e.g., sorting and others)
- ❖ Higher-level code can pin and unpin a frame
 - Pin: I need to work on this frame in memory
 - Unpin: I no longer need this frame
 - A completely unpinned frame is a candidate for replacement
 - ☞ In some systems you can hate a frame (i.e., suggesting it for replacement)
- ❖ A frame becomes dirty when it is modified
 - Only dirty frames need to be written back to disk
 - ☞ Related to transaction processing (more on it later in the semester)

Standard OS replacement policies

- ❖ Example
 - Current buffer pool: 0, 1, 2
 - Past requests: 0, 1, 2
 - Incoming requests: 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, ...
 - ☞ Which frame to replace?
- ❖ Optimal: replace the frame that will not be used for the longest time (2)
- ❖ Random (0, 1, or 2 with equal probability)
- ❖ LRU: least recently used (0)
- ❖ LRU approximation: clock, aging
- ❖ MRU: most recently used (2)

Problems with OS buffer management

Stonebraker. "Operating System Support for Database Management." *CACM*, 1981 (in red book).

- ❖ Performance problems
 - Getting a page from the OS to user space is usually a system call (process switch) and copy
- ❖ Replacement policy
 - LRU, clock, etc. often ineffective
 - DBMS knows access pattern in advance and therefore should dictate policy → major OS/DBMS distinction
- ❖ Prefetch policy
 - DBMS knows of multiple “orders” for a set of records; OS only knows physical order
- ❖ Crash recovery
 - DBMS needs more control

Next

7

Chou and DeWitt. "An Evaluation of Buffer Management Strategies for Relational Database Systems." *VLDB* 1985 (in red book).

- ❖ Old algorithms
 - Domain separation algorithm
 - "New" algorithm
 - Hot set algorithm
- ❖ Query locality set model
- ❖ DBMIN algorithm

Domain separation algorithm

8

- ❖ Split work/memory into domains; LRU within each domain; borrow from other domains when out of frames
 - Example: one domain for each level of the B⁺-tree
- ❖ Limitations
 - Assignment of pages to domains is static, and ignores how pages are used
 - Example: A data page is accessed only once in a scan, but the same data page is accessed many times in a NLJ
 - Does not differentiate relative importance between types of pages
 - Example: An index page is more important than a data page
 - Memory allocation is based on data rather queries → need orthogonal load control to prevent thrashing

The "new" algorithm

9

- ☞ Observations based on the reference patterns of queries
 - Priority is not a property of a data page, but of a relation
 - Each relation needs a "working set"
- ❖ Divide buffer pool into chunks, one per relation
- ❖ Prioritize relations according to how often their pages are reused
- ❖ Replace a frame from the least reused relation and add it to the chunk of the referenced relation
- ❖ Each active relation is guaranteed with one frame
- ❖ MRU within each chunk (seems arbitrary)
- ❖ Simulation look good; implementation did not beat LRU

Hot set algorithm

10

- ☞ Exploit query behavior more!
- ❖ A set of pages that are accessed over and over form a hot set
 - "Hot points" in the graph of buffer size vs. number of page faults
 - Example: For nested-loop join $R \bowtie S$, size of hot set is $B(S) + 1$ (under LRU)
- ❖ Each query is given enough memory for its hot set
- ❖ Admission control: Do not let a query into the system unless its hot set fits in memory
- ❖ Replacement: LRU within each hot set (seems arbitrary)
- ❖ Derivation of hot set assumes LRU, which may be suboptimal
 - Example: What is better for nested-loop join?

Query locality set model

11

- ❖ Observations
 - DBMS supports a limited set of operations
 - Reference patterns are regular and predictable
 - Reference patterns can be decomposed into simple patterns
- ❖ Reference pattern classification
 - Sequential
 - Random
 - Hierarchical

Sequential reference patterns

12

- ❖ Straight sequential: read something sequentially once
 - Example: selection on unordered table
 - ☞ Each page is only touched once, so just buffer one page
- ❖ Clustered sequential: repeatedly read a "chunk" sequentially
 - Example: merge join; rows with the same join column value are scanned multiple times
 - ☞ Keep all pages in the chunk in buffer
- ❖ Looping sequential: repeatedly read something sequentially
 - Example: nested-loop join
 - ☞ Keep as many pages as possible in buffer, with MRU replacement

Random reference patterns

13

- ❖ Independent random: truly random accesses
 - Example: index scan through a non-clustered (e.g., secondary) index yields random data page access
 - ☞ The larger the buffer the better?
- ❖ Clustered random: random accesses that happen to demonstrate some locality
 - Example: in an index nested-loop join, inner index is non-clustered and non-unique, while outer table is clustered and non-unique
 - ☞ Try to keep in buffer data pages of the inner table accessed in one cluster

Hierarchical reference patterns

14

- ❖ Example: operations on tree indexes
- ❖ Straight hierarchical: regular root-to-leaf traversal
- ❖ Hierarchical with straight sequential: traversal followed by straight sequential on leaves
- ❖ Hierarchical with clustered sequential: traversal followed by clustered sequential on leaves
- ❖ Looping hierarchical: repeatedly traverse an index
 - Example: index nested-loop join
 - ☞ Keep the root index page in buffer

DBMIN algorithm

15

- ❖ Associate a chunk of memory with each file instance (each table in FROM)
 - This chunk is called the file instance's locality set
 - Instances of the same table may share buffered pages
 - But each locality set has its own replacement policy
 - ☞ Based on how query processing uses each relation (finally!)
 - ☞ No single policy for all pages accessed by a query
 - ☞ No single policy for all pages in a table
- ❖ Estimate locality set sizes by examining the query plan and database statistics
- ❖ Admission control: a query is allowed to run if its locality sets fit in free frames

DBMIN algorithm (cont'd)

16

- ❖ Locality sets: each "owns" a set of pages, up to a limit l
- ❖ Global free list: set of "orphan" pages
- ❖ Global table: allow sharing among concurrent queries
- ❖ Query q requests page p
 - If p is in memory and in q 's locality set
 - Just update usage statistics of p
 - If p is in memory and in some other query's locality set
 - Just make p available to q ; no further action is required
 - If p is in memory and in the global free list
 - Add p to q 's locality set; if q 's locality set exceeds its size limit, replace a page (release it back to the global free list)
 - If p is not in memory
 - Use a buffer from global free list to get p in; proceed as in the previous case

Locality sets for various ref. patterns

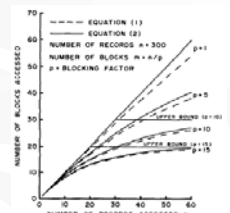
17

- ❖ Straight sequential
 - Size = 1
 - Just replace as needed
- ❖ Clustered sequential
 - Size = number of pages in the largest cluster
 - FIFO or LRU
- ❖ Looping sequential
 - Size = number of pages in the table
 - MRU

Locality sets for more ref. patterns

18

- ❖ Independent random
 - Size = 1 (if odds of revisit is low), or b (expected number of block accessed by a given number k of random record accesses; Yao, 1977)
 - Use $(k - b) / b$ to choose between 1 and b
 - Replacement policy does not matter
- ❖ Clustered random
 - Size = number of blocks in the largest cluster (\approx number of tuples because of random access, or use Yao's formula)
 - LRU or FIFO (why?)



Locality sets for more ref. patterns

19

- ❖ Straight hierarchical, hierarchical/straight sequential: just like straight sequential
 - Size = 1
 - Just replace as needed
- ❖ Hierarchical/clustered sequential: like clustered sequential
 - Size = number of index pages in the largest cluster
 - FIFO or LRU
- ❖ Looping hierarchical
 - At each level of the index you have random access among pages
 - Use Yao's formula to figure out how many pages need to be accessed at each level
 - Size = sum over all levels that you choose to worry about
 - LIFO with 3-4 buffers should be okay

Simulation study

20

- ❖ Hybrid simulation model
 - Trace-driven simulation
 - Recorded from a real system (running Wisconsin Benchmark)
 - For each query, record its execution trace
 - Page read/write, file open/close, etc.
 - Distribution-driven simulation
 - Generated by some stochastic model
 - Synthesize the workload by merging query execution traces
- ❖ Simulator models CPU, memory, and one disk
- ❖ Performance metric: query throughput

Workload

21

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 #	Query Operators	Selectivity	Access Path of Selection	Join Method	Access Path of Join
I	select(A)	1%	clustered index	-	-
II	select(B)	1%	non-clustered index	-	-
III	select(A) join B	2%	clustered index	index join	clustered index on B
IV	select(A) join B	10%	sequential scan	index join	non-clustered index on B
V	select(A) join B*	3%	clustered index	nested loops	sequential scan over B*
VI	select(A) join A*	4%	clustered index	hash join	hash on result of select(A)

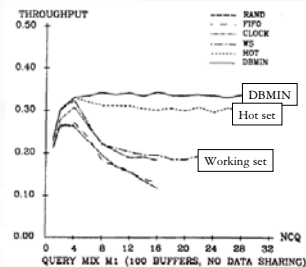
A, B: 10K tuples; A*: 1K tuples; B*: 300 tuples; 182 bytes per tuple.

Description of Base Queries

- ❖ Mix 1: all six types equally likely
- ❖ Mix 2: I and II together appear 50% of the time
- ❖ Mix 3: I and II together appear 75% of the time

Mix 1 (no data sharing)

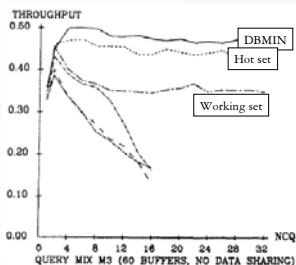
22



- ❖ Thrashing is evident for simple algorithms with no load control
- ❖ Working set (a popular OS choice) fails to capture join loops for queries with high memory demand (types V and VI)
 - It still functions (though suboptimally) with large number of current queries (NCQ)

Mix 3 (no data sharing)

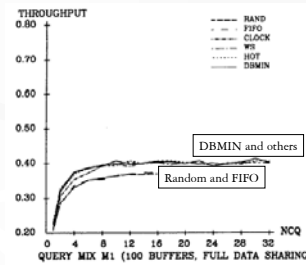
23



- ❖ Thrashing is still evident
- ❖ Working set fares better because mix 3 has more simple queries and fewer ones of types V and VI

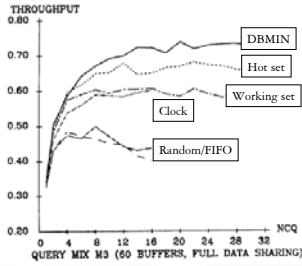
Mix 1 (full data sharing)

24



- ❖ With full data sharing, locality is easier to capture
 - Performance improves across the board and the gap disappears
 - Random and FIFO do not capture locality as effectively as others

Mix 3 (full data sharing)

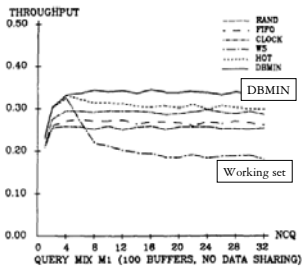


- ❖ Performance starts to diverge again
 - Mix 3 is dominated by lots of small queries, and locality becomes harder to capture

Load control

- ❖ Mechanism to check resource usage in order to prevent system from overloading
- ❖ Rule of thumb: “50% rule”—keep the paging device busy half of the time
- ❖ Implementation
 - Estimator measures the utilization of device
 - Optimizer analyzes measurements and decides whether/what load adjustment is appropriate
 - Control switch activates/deactivates processes according to optimizer’s decisions

Mix 1 (load control, no data sharing)



- ❖ DBMIN still the best
- ❖ (Simple algorithms + load control) outperforms working set!
- ❖ Cons of load control
 - Runtime overhead
 - Non-predictive
 - Only responds after undesirable condition occurs

Conclusion

- ❖ Same basic access patterns come up again and again in query processing
- ❖ Make buffer manager aware of these access patterns
- ☞ Look at the workload, not just the content
 - Contents can at best offer guesses at likely workloads