



# Operating System Support for Database Management & Buffer Management Strategies for Database Systems

---

Ioannis Roussos



# Operating System Support for Database Management

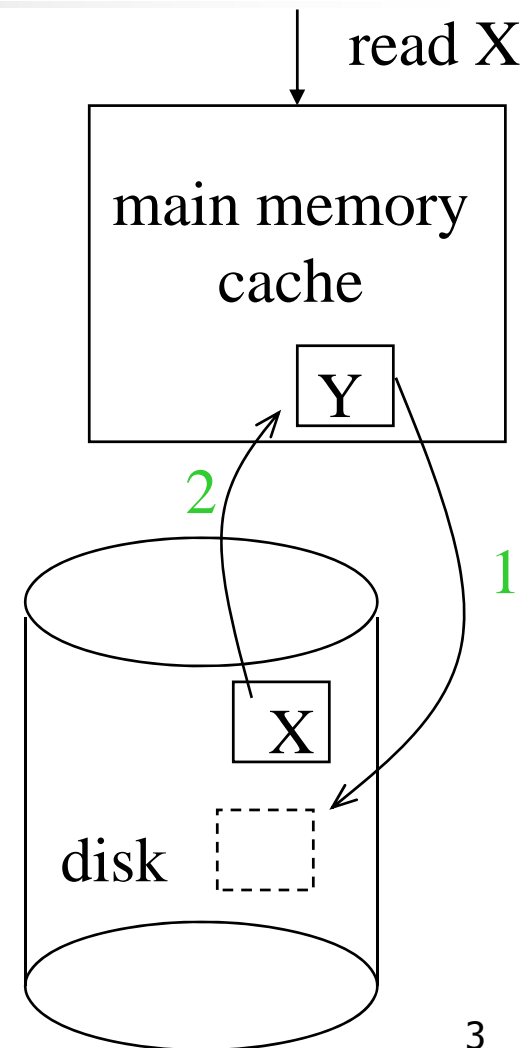
---

- Operating system services in many systems are either *too slow* or *inappropriate* for DBMS.
- Examples from UNIX, drawn from the experience of *Stonebraker* during the design and implementation of the *INGRES* database system.
  - Buffer pool management.
  - File system.
  - Scheduling, process management, and IPC.
  - Consistency control.
- Most of the points have general applicability to other operating systems.

# OS Support for DBMS

## *Buffer Pool Management*

- Main memory cache for the file system.
- The OS establishes policies in each of the following areas:
  - *Fetch Policy*
  - Placement Policy
  - *Replacement Policy*
  - Resident Set Management
  - Cleaning Policy
  - Load Control
- Unix buffer manager:
  - *LRU* replacement strategy
  - Fetch Policy: Prefetch when sequential access to a file is detected





# OS Support for DBMS

## *Buffer Pool Management*

---

### *Performance Problems*

- Fetching a block from disk →  
system call + core-to-core move
- Cost(fetch 512 bytes) > 5000 instructions
- Cost(fetch 1 byte from buffer pool) ≈ 1800 instructions
- DBMS are forced to put a buffer pool in user space to reduce overhead.
- ✓ Requirement from Operating Systems
  - cut the access overhead to a *few hundred* instructions.

# OS Support for DBMS

## *Buffer Pool Management*

### *LRU Replacement*

(toss immediately)

- Good for *general* buffer management (locality,...)

- Database access pattern in INGRES

1. Seq. access to blocks which will not be rereferenced.

(MRU) 2. Seq. access to blocks which will be cyclically rereferenced.

3. Random access to blocks which will not be rereferenced.

(LRU) 4. Random access to blocks which will possibly be rereferenced.

✓ Requirement from Operating Systems:

- Interface that will allow it to *accept "advice"* from an application (e.g., a DBMS) concerning the replacement strategy.



# OS Support for DBMS

## *Buffer Pool Management*

---

### *Fetch policy*

- Unix correctly prefetches pages when sequential access is detected.
- Failure to predict DB access patterns:
  - Logical order in DB patterns  $\neq$  physical order
- Except in rare cases, the DBMS knows from the beginning of the examination of a block which block will access next (not necessarily the next in logical file order). → There is no way for an OS to implement the correct prefetch strategy.
- ✓ Requirement from Operating Systems:
  - *Prefetch advice facility.*

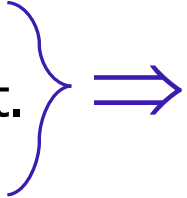


# OS Support for DBMS

## *Buffer Pool Management*

---

### *Crash Recovery*

1. Maintain an intentions list.
  2. When the intentions list is complete, a commit flag is set.
  3. Process the intentions list making the actual updates.
- The page in which the commit flag exists must be forced to disk *after* all pages in the intentions list.
- ✓ Requirement from Operating Systems:
- A *selected force out* service, which would push the intentions list and the commit flag to disk in the proper order.
- 



# OS Support for DBMS *File System*

---

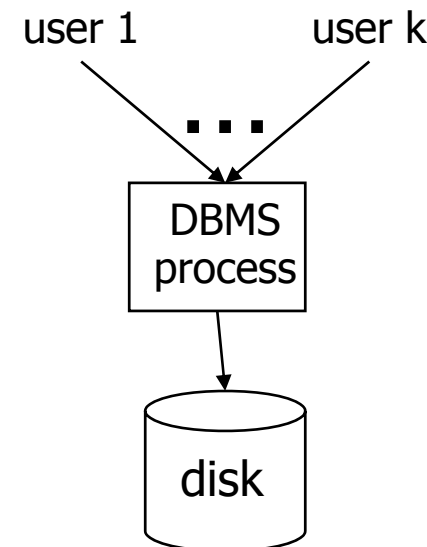
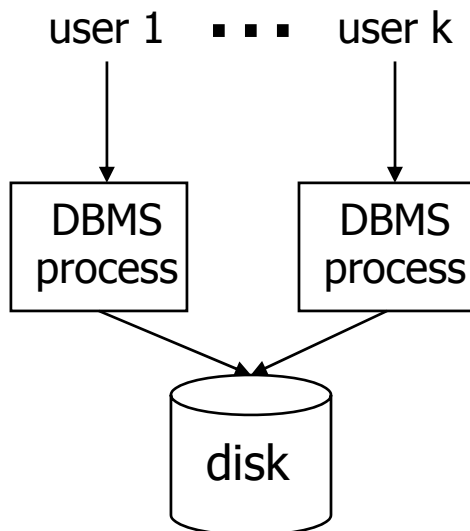
- Unix files are *character arrays* of dynamically varying size. These character arrays can usually be expanded *one block at a time*:
  - blocks of a given file are scattered over a disk volume → the next logical block in a file is *not* necessarily physically close to the previous one.
- Desired service: A way to store data in contiguous physical blocks → *extent based* file system.
- Unix provides 2 tree structures:
  - blocks in a given file
  - the file in a given file system.
  - DBMS use a 3<sup>rd</sup> for keyed access.
- Extra overhead from 3 separately managed trees.

# OS Support for DBMS

## *Process Management*

Two ways to organize a *multiuser* database system:

- Each concurrent user runs in a *separate process*.
- Buffer pool and lock table should be *global* (in a shared segment).
- Unique database process which acts as a *server*.
- Concurrent users send messages to server with work requests.





# OS Support for DBMS

## *Process Management*

---

### *Process per user approach*

- Every time a database process issues an I/O request → task switch.
  - In many OS: task switch cost > 1000 instructions.
- What happens if the OS scheduler deschedules a process while it is holding a lock for a critical section (e.g. the root of an index)?
  - All other processes cannot execute very long without accessing the buffer pool → queue up behind the locked resource (convoy phenomenon)
- ✓ Requirement from Operating Systems:
  - Create a special scheduling class for the DBMS. Processes in this class would *never* be forcibly descheduled unless they *voluntarily* relinquish the CPU.
  - Fast path through the task switch/scheduler loop to pass control to an other special process.



# OS Support for DBMS

## *Process Management*

---

### *Server Model*

- The server must do its own scheduling and multitasking → painful duplication of OS facilities.
- One message per I/O request, instead of a task switch per I/O. Although it sounds strange:
  - A message is in many OS more expensive than a task switch.
  - In most OS, the cost for a round-trip message is several thousand instructions (in Unix, cost  $\approx$  5000 instr.).
- ✓ Requirement from Operating Systems:
  - Create a cheaper message system (at least for DBMS consumers).



# OS Support for DBMS

## *Consistency Control*

---

- Lock objects for shared or exclusive access.
  - OS provide services for locking *files*.
  - Very few provide finer granularity locks.
  - Locks on *pages* or *records* are essential in most DBMS.
- It has been suggested that both concurrency control and crash recovery for transactions be provided entirely inside the operating system.
  - Only problem: If *Buffer Management* is provided by the DBMS  
→ In order to solve conflicts with the operating system's transaction management, operating system facilities are being duplicated (e.g. commit point).
  - Solution: Buffering, concurrency control and crash recovery must all be provided by the operating system **OR** the DBMS.

# OS Support for DBMS

## *Consistency Control*

### Ordering Dependencies

- Consider the following employee data:

<u>Empname</u>	<u>Salary</u>	<u>Manager</u>
Smith	10000	Brown
Jones	9000	None
Brown	11000	Jones

and the update which gives a 20% pay cut to all employees who earn more than their managers.

- Presumably, Brown will be the only employee to receive a decrease (although there exist alternative semantic definitions).
- Suppose that the DBMS updates the data set as it finds overpaid employees, depending on the OS to provide backout or recover-forward on crashes → the outcome of the update depends on the order of execution.
- If the OS maintains both the buffer pool *and* the intentions list for crash recovery, it can avoid this problem.



# Buffer Management Strategies for Database Systems

---

- Chou & DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems" *VLDB'85*
  - Old algorithms for DBMS Buffer Management.
  - Query Locality Set Model (QLSM).
  - DBMIN.
  - Evaluation.
- *Motivation:*
  - Stonebraker's paper.
  - Hot set algorithm → inconclusive simulation results.



# Buffer Management for DBMS

## *Domain Separation*

---

- Motivation: indexes.
- Pages are classified into types. Each type is *separately* managed in a different domain of buffers.
- When a page of certain type is needed, a buffer is allocated from corresponding domain.
  - *LRU* within domain.
  - If no free pages inside domain(e.g. all locked for I/O) → borrow from an other domain.
- 8-10% improvement in throughput compared to LRU.



# Buffer Management for DBMS

## *Domain Separation*

---

### *Limitations:*

- Domains are *static*. The importance of a page may vary in different queries: Keep a page when accessed in a nested loops join, but discard it when accessed in a sequential scan.
- Does not differentiate the relative importance between different types of pages.
- Partitioning buffers according to domains, rather than queries  
→ Interference among competing users.

### Several extensions:

- Group LRU (GLRU): fixed priority ranking among domains.
- Effelsberg: dynamically vary the size of each domain (working-set like partitioning scheme).



# Buffer Management for DBMS

## *"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.
- Algorithm:
  - Each active relation is assigned a resident set which is initially empty.
  - The resident sets of relations are linked in a priority list. A relation is near top, if its pages are unlikely to be reused.
  - Ordering of relation is pre-determined, and maybe adjusted subsequently.
  - When a page fault occurs: Search from top of the list
  - Within each relation: *MRU*. Each relation has one active buffer, which is exempt from replacement consideration.



# Buffer Management for DBMS

## *"New" Algorithm*

---

### *Pros*

- A new approach that tracks the locality of a query through relations

### *Weak points*

- 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
- Results from implementation for INGRES: FAILED to improve performance (compared to existing LRU)

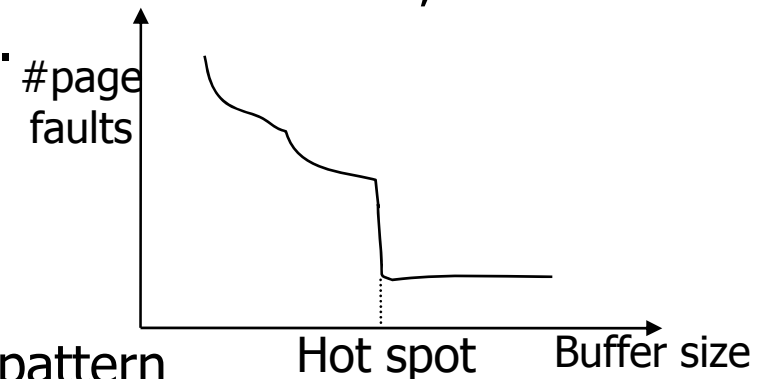
# Buffer Management for DBMS

## *Hot Set Algorithm*

- Query behavior model, that integrates advance knowledge on reference patterns into the model.
- *Hot set* : a set of pages over which there is a looping behavior.
- Give to the query a buffer large enough to hold the hot sets.
- Plotting the #page faults as a function of buffer size, we can observe discontinuities (*hot spots*).
- Example: nested loops join

*Hot spot = #pages in inner relation + 1*

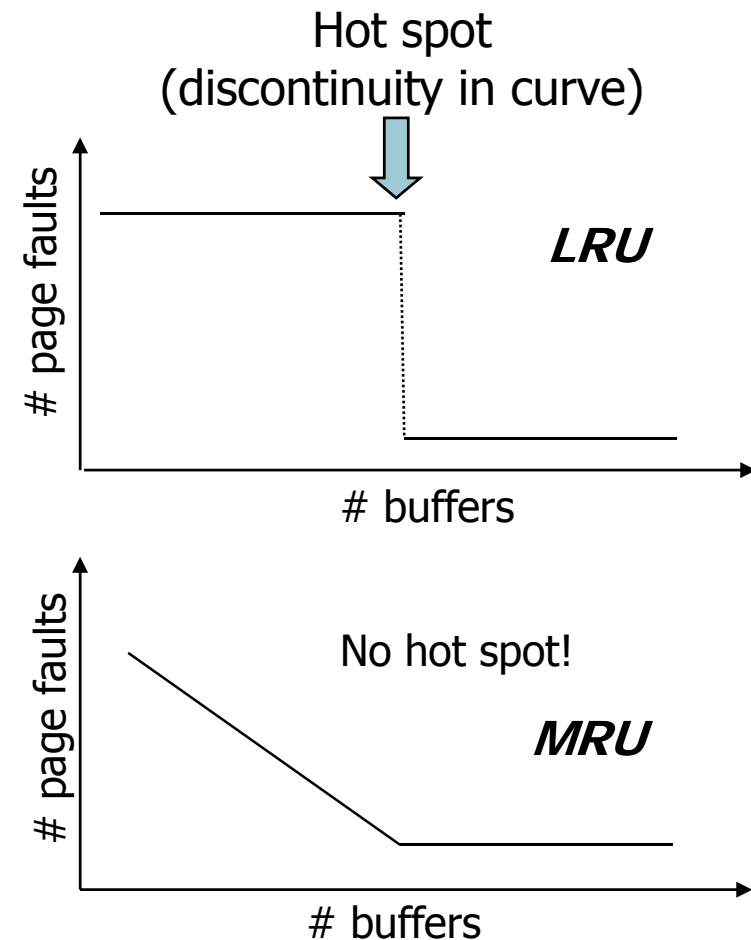
- Provides more accurate reference pattern than a stochastic model.



# Buffer Management for DBMS

## *Hot Set Algorithm*

- Key ideas
  - Give query |hot set| pages
  - Hot set size computed by query optimizer
  - Use LRU within each partition
- Problems
  - Based on LRU, which is inappropriate for certain looping behavior (MRU is better).
  - So, does not truly reflect the inherent behavior of some reference patterns, but the behavior under LRU.
  - Over-allocates pages for some phases of query.



# Buffer Management for DBMS

## *QLSM*

- Like the hot set model, integrates advance knowledge on reference patterns into the model.
- Avoids the potential problems by separating the modeling of reference behavior from any particular buffer management algorithm.
- Intuition:
  - DBMS support a *limited set* of operations.
  - Pattern of page references exhibited by these operations are *regular* and *predictable*.
  - Each reference pattern can be decomposed into the composition of a number of simple reference patterns.
- Reference pattern classification:
  - Sequential
  - Random
  - Hierarchical

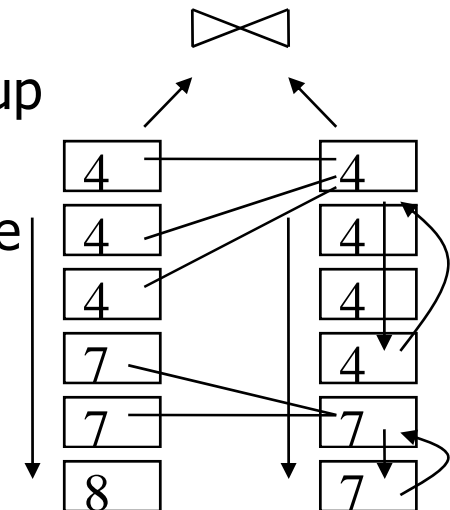
# Buffer Management for DBMS

## QLSM

- *Straight sequential (SS)*
  - File scan without repetition
    - E.g., selection on an unordered relation
  - Buffer space: 1 page.
- *Clustered sequential (CS)*
  - Local rescan in a SS scan: A scan may back-up a sort distance and then start forward.
  - Merge-join (sequential), with records with the same key in the inner relation
  - Buffer space: #pages in largest cluster
  - Replacement algorithm: FIFO/LRU

Table R

R1
R2
R3
R4
R5
R6



# Buffer Management for DBMS

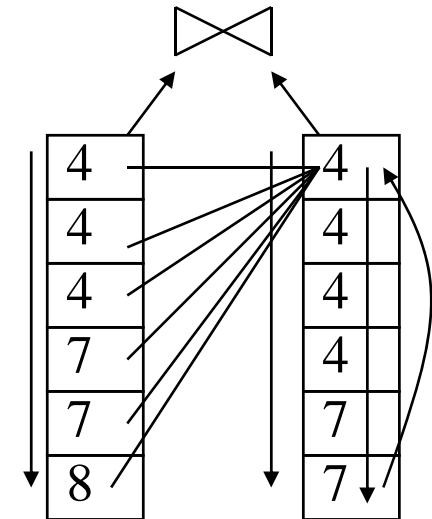
## *QLSM*

### ■ *Looping sequential (LS)*

- Sequential reference be repeated several times
  - e.g., inner S for nested-loop-join
- Buffer space: As many pages as possible
- Replacement algorithm: MRU

### ■ *Independent Random (IR)*

- Independent random accesses
  - e.g., non-clustered index scan
- Buffer space: Use Yao's formula, which estimates the total number of pages referenced **b** in a series of **k** random accesses. If page references are sparse, use 1 page, else use b pages: Define  $r = (k-b)/b$  the *residual value* of a page. if  $r < \beta$  use 1 page, else use b pages.



# Buffer Management for DBMS

## *QLSM*

- *Clustered Random (CR)*
  - Random accesses which demonstrate locality
    - e.g., join where:  
Inner relation with non-clustered, non-unique index on join column.  
Outer relation is a clustered file with non-unique keys.
  - Buffer space: #pages in largest cluster.
  - As in CS.
- *Straight Hierarchical (SH)*
  - Access index pages ONCE (retrieve a single tuple).
  - Buffer space: 1 page.
  - Followed by straight sequential scan (*H/SS*)
    - Like SS
  - Followed by clustered scan (*H/CS*)
    - Like CS

# Buffer Management for DBMS

## *QLSM*

- *Looping Hierarchical (LH)*

- Repeatedly traverse an index
  - e.g., when inner index in join is repeatedly accessed
- Pages closer to the root are more likely to be accessed than those closer to the leaves.
  - Consider a full tree of height **h** and with fan-out factor **f**.
  - During the traversal of the tree, at a level *i*: one out of the  $f^i$  pages is referenced

- Use the *residual value* defined at IR: Let  $b_i$  be the number of pages accessed at level *i* as estimated by Yao's formula. Buffer size:

$$\left( 1 + \sum_{i=1}^j b_i \right) + 1$$

where *j* is the largest *i* such that  $\frac{k - b_i}{b_i} > \beta$

- Replacement algorithm: LIFO (in order to keep the root).



# Buffer Management for DBMS

## *DBMIN*

---

- Buffer management algorithm based on the QLSM.
- Buffer management on a *per file instance* basis.
- Each file instance has a *locality set*: the set of buffers associated with that file instance.
- Manage each locality set:
  - Separately.
  - According to the intended usage of the file instance (access pattern that will chose).
- If a buffer contains a page that does not belong to any locality set, the buffer is placed on a *global free list*.
- A page in the buffer can belong to at most one locality set.



# Buffer Management for DBMS

## *DBMIN*

---

### *Parameters*

- $N$  : total number of buffers(page frames) in the system.
- $l_{ij}$  : max number of buffers that can be allocated to file instance  $j$  of query  $i$  (desired size).
- $r_{ij}$  : number of buffers allocated for file instance  $j$  of query  $i$  (actual size).



# Buffer Management for DBMS

## *DBMIN*

---

### *Algorithm*

- Initialize the global table and link all the buffers in the system on global free list.
- When a file is opened, its associated locality set size and replacement policy are given to the buffer manager:
  - An empty locality set is initialized for the file instance.
  - $r$  is initialized to 0.
  - $l$  is initialized to the given locality set size.
- When a page is requested by a query, a search is made to the global table, followed by an adjustment to the associated locality set.

# Buffer Management for DBMS

## *DBMIN*

### *Algorithm (cont.)*

Three possible cases:

1. The page is found in both the global table and the locality set:
  - Only the usage statistics need to be updated.
2. The page is found in memory but not in the locality set:
  - If the page already has an owner, the page is simply given to the requesting query and no further actions are required.
  - Otherwise,
    - the page is added to the locality set.
    - increment  $r$ . If  $r > l$ , a page is chosen and released back to the global free list according to the local replacement policy.
    - usage statistics are updated.
3. The page is not in memory:
  - Bring the page from disk into a buffer in global table. Proceed as in case 2.

# Buffer Management for DBMS

## *DBMIN*

### *Algorithm (cont.)*

- The *load controller* is activated when a file is opened or closed. Immediately after a file is opened, the load controller checks whether:  $\sum_i \sum_j l_{ij} < N$   
for all active queries  $i$  and their file instances  $j$ .
  - If so, the query is allowed to proceed;
  - Otherwise, it is suspended and placed at the front of a waiting queue.
- Local replacement policy and estimated size for the locality set of each file instance → The ones of the QLSM model.

# Buffer Management for DBMS

## *Evaluation of algorithms*

### *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**

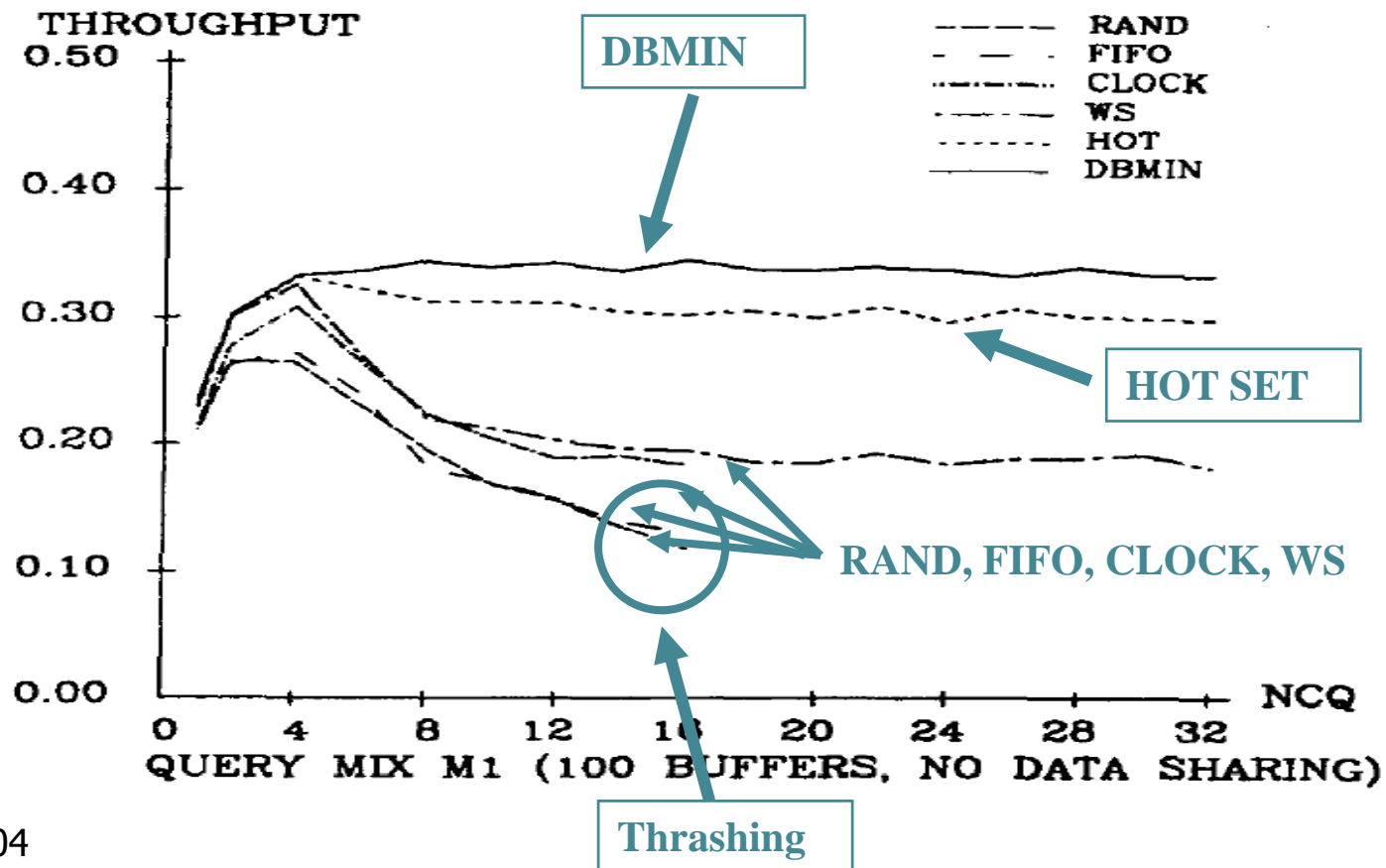
Query Mix	Type I	Type II	Type III	Type Iv	Type V	Type VI
M1	16.67	16.67	16.67	16.67	16.67	16.67
M2	25.00	25.00	12.50	12.50	12.50	12.50
M3	37.5	37.5	6.25	6.25	6.25	6.25

(in %)

# Buffer Management for DBMS

## *Evaluation of algorithms*

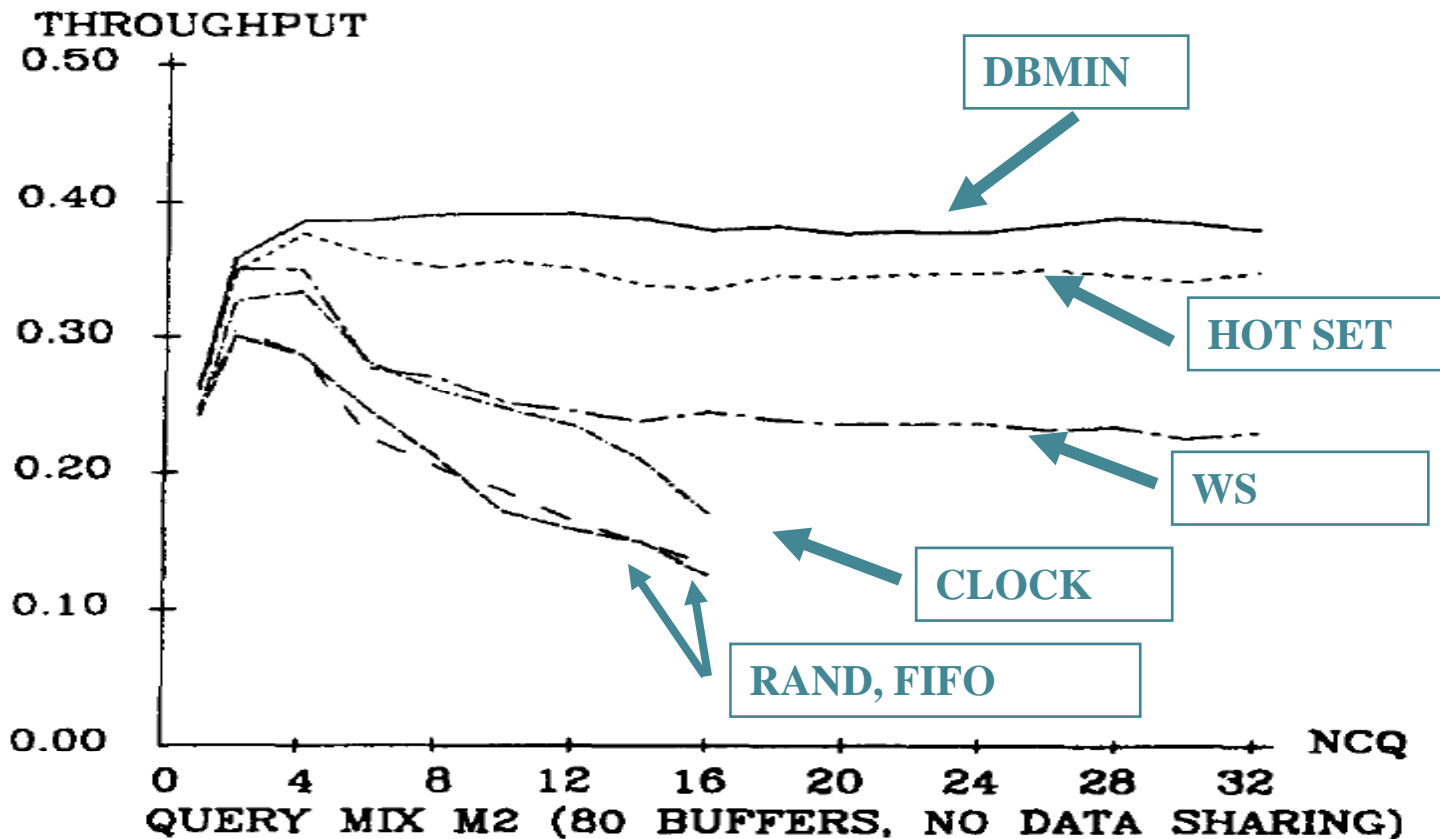
*Query Mix One: all queries equally distributed*



# Buffer Management for DBMS

## *Evaluation of algorithms*

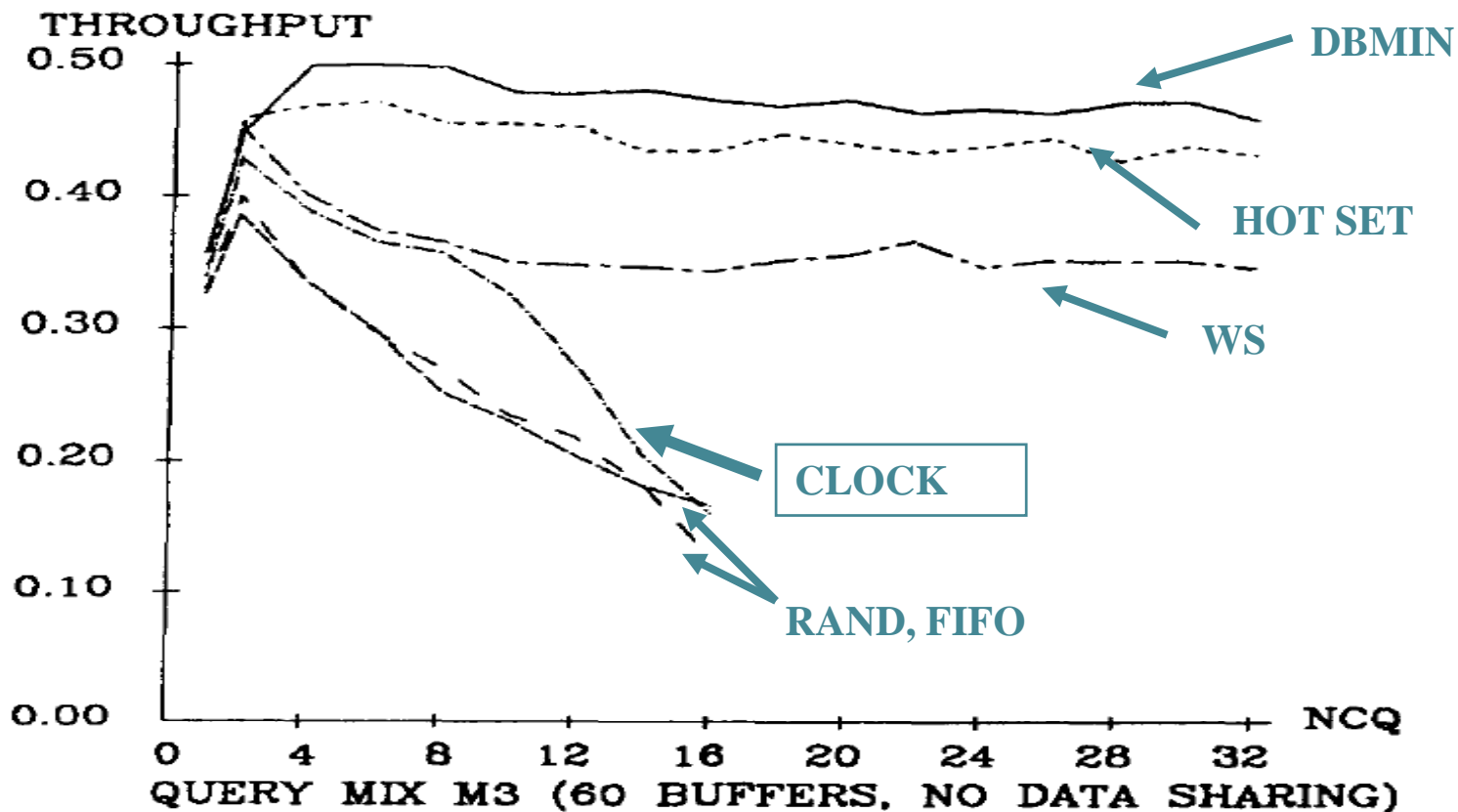
*Query Mix Two: I + II = 50%*



# Buffer Management for DBMS

## *Evaluation of algorithms*

*Query Mix Three: I + II = 75%*





# *References*

---

- Stonebraker, "Operating System Support for Database Management", CACM 1981
- Chou, Dewit, "An Evaluation of Buffer Management Strategies for Relational Database Systems", VLDB 1985
- Effelsberg, "Principles of database buffer management", TODS 1984
- Sacco, "Buffer management in relational database systems", TODS 1986