

SQL Continued (Chapter 5)

Sailors(*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

Boats(*bid*: integer, *bname*: string, *color*: string)

Reserves(*sid*: integer, *bid*: integer, *day*: date)

Find the names of sailors who reserved all boats:

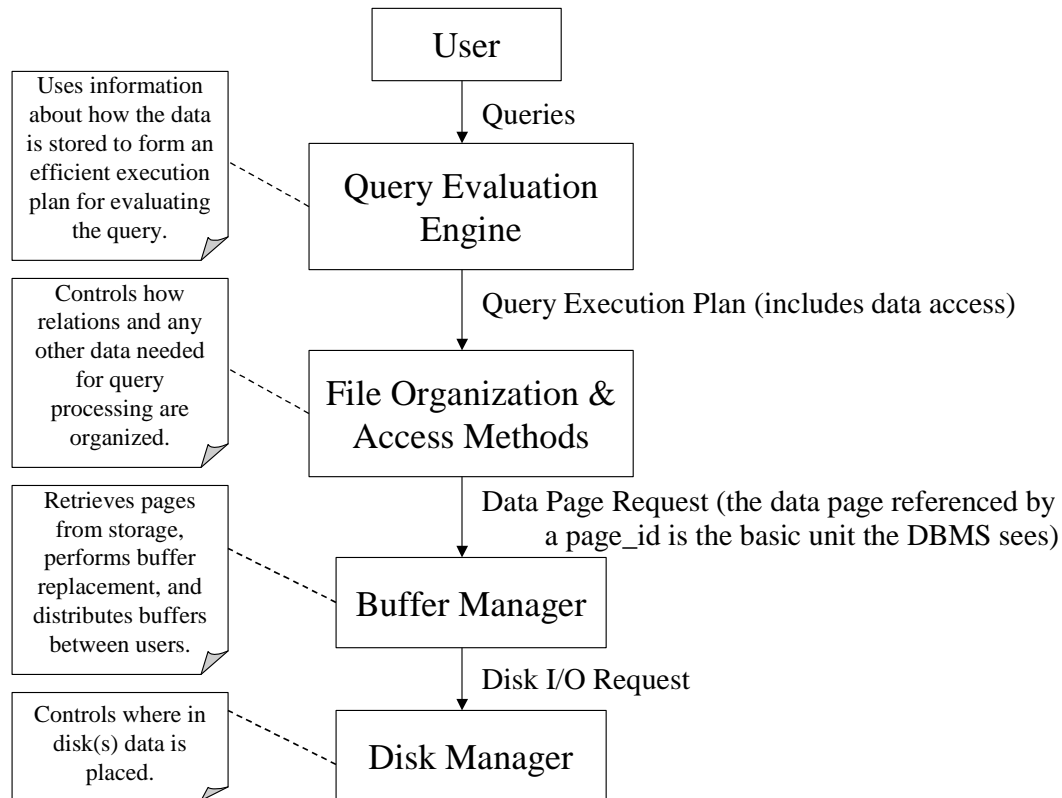
Solution 3: Group the reservations by sailor and choose those sailors with a count of unique boats reserved equal to the total number of boats in the Boats table. (see previous class for first two solutions to this problem)

```
SELECT      S.sname
FROM        Reserves R, Sailors S
WHERE       S.sid = R.sid
GROUP BY   R.sid, S.sname
HAVING      COUNT(DISTINCT bid) = (
                                SELECT COUNT(B.bid)
                                FROM Boats B
                                );
```

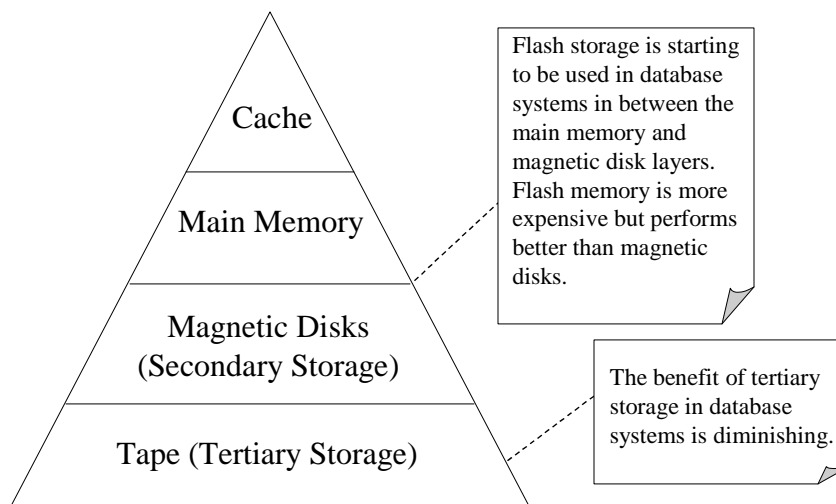
Note: Grouping by R.sid alone would be incorrect since the attribute list from the SELECT clause must always be a subset of the group list.

Data Storage in DBMS (Chapter 9)

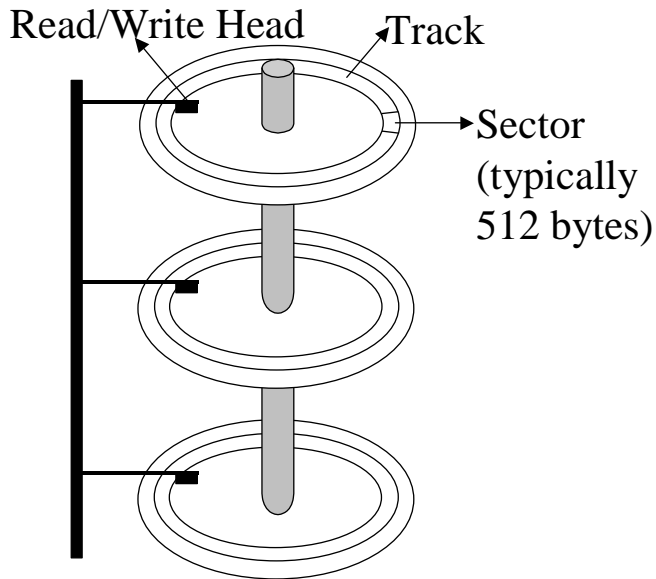
Structure of a DBMS:



1. Memory Storage Pyramid:



2. Disks:



- A disk consists of multiple plates stacked together.
- To read a sector, the head moves right or left to position itself over the desired track. It then waits for the desired sector to pass underneath it.

Cost Model:

1. Seek time (time for head to move into position) is in the 10s of milliseconds.
2. Rotational latency (time for sector to rotate into position) is in the 10s of milliseconds.
3. Data transfer time is very fast. 1 & 2 are the bottlenecks.

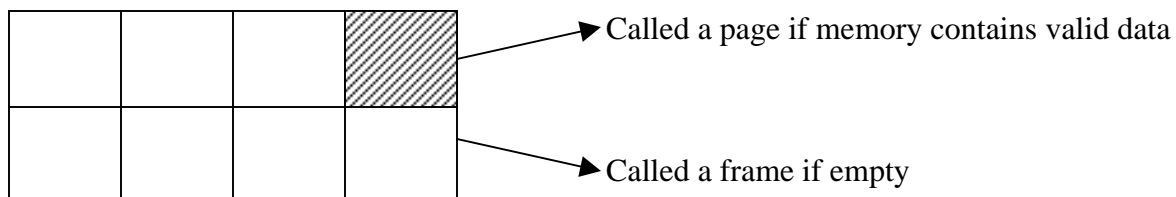
In general, a sequential access data pattern is preferred over random access since it will reduce the overhead of seek time and rotational latency.

3. Pages:

A page is a consecutive chunk of sectors on a disk (typically 4K – 16K).

4. Buffer Pool:

The DBMS requests the buffer pool as one large chunk of memory from the OS. The DBMS then divides the memory into page sized chunks.



A page contains multiple records. A request to the buffer manager for a record consists of a record id: <page_id, record_#>

The page_id identifies the page a record is stored in and the record_# specifies the offset within that page. When a record is requested, the buffer manager will always pull in the entire page a record is stored in.

If page_id is already in memory, the buffer manager returns a pointer to the page.

Otherwise, if there are empty frames available, the buffer manager chooses a frame, copies the page from disk and returns a pointer to the page.

Otherwise, the buffer manager must perform buffer replacement using some replacement strategy (i.e. LRU).

To manage the buffer pool, the DBMS must maintain additional structures for each page:

pin_count = Number of queries currently using the page (incremented when a process requests a page and decremented when the process releases it).

dirty_bit = The data has been modified and must be flushed to disk before this page is replaced.

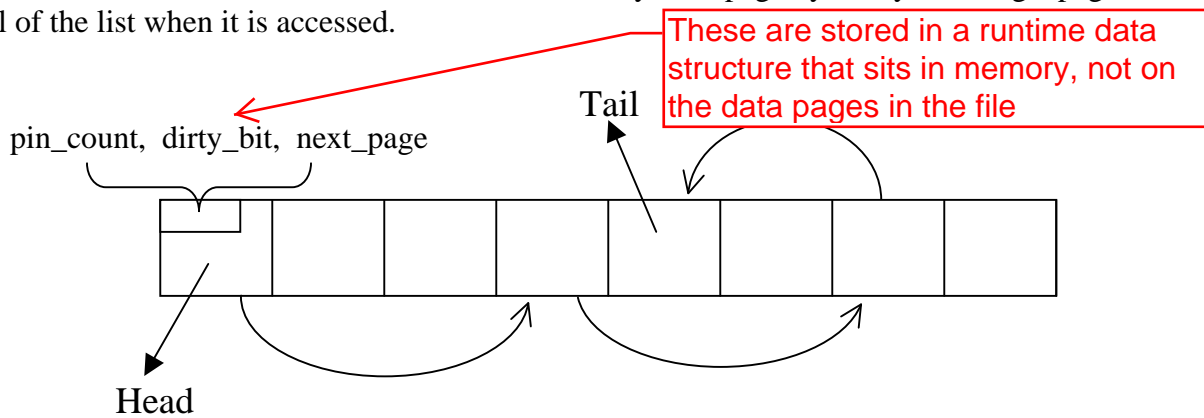
4. Page Replacement:

LRU is a popular page replacement strategy.

How do we implement LRU and what are the performance considerations?

A linked list can be used to keep track of the pages with pin_count equal to 0. These are the pages that can be safely replaced since no processes are currently using them.

The linked list can also be used to track the least recently used page by always moving a page to the tail of the list when it is accessed.



To get a new page: Retrieve the head (this is the least recently used page).

For each page access: Move the accessed page to the tail.

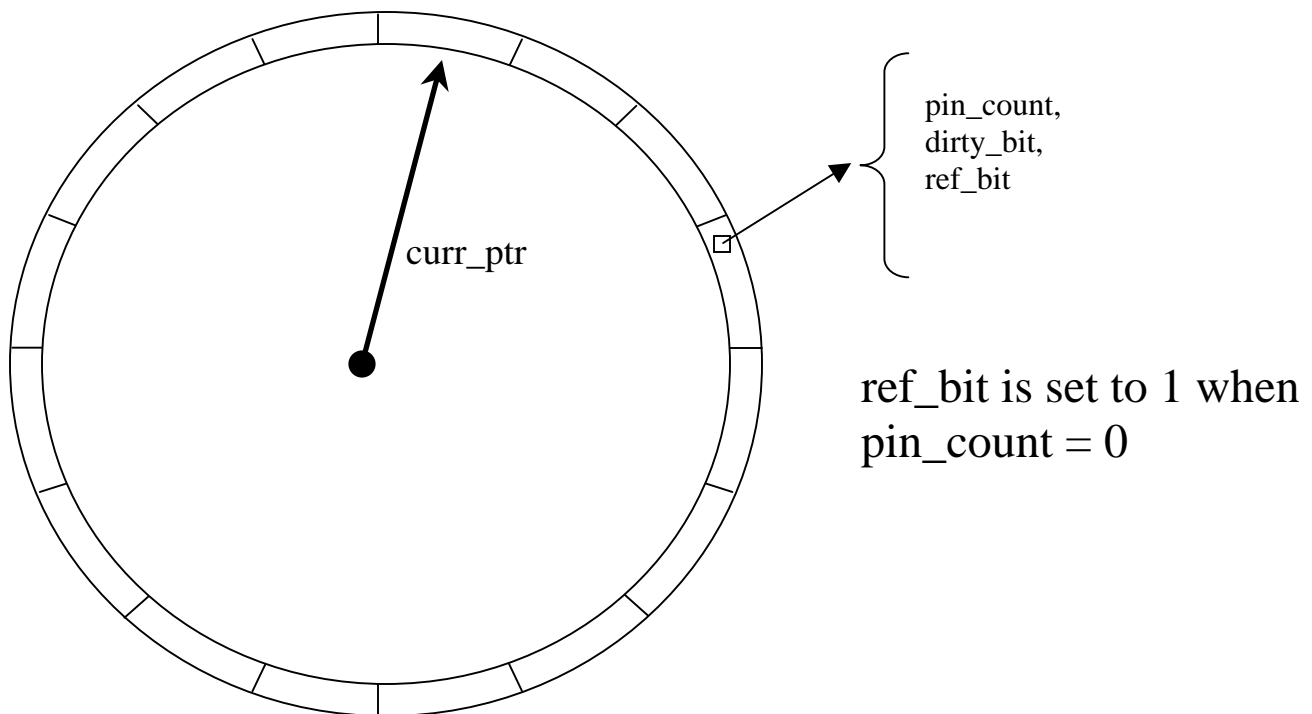
LRU has been shown to be close to optimal (in terms of hit rate) unless we have some special types of access patterns.

Problems of LRU:

1. Overhead. The very common case of accessing a page requires an update to the linked list.
2. Poor performance in special access patterns.

For the first case, we can use the clock replacement policy instead of LRU. Clock replacement simulates the success rate of LRU with lower overhead.

All memory pages are placed into a circular queue:



Case 0: If clock is pointing to a page with $pin_count > 0$, it just moves on to the next page:

```
curr_ptr++;
```

Case 1: If clock is pointing to a page with $pin_count = 0$ and $ref_bit = 1$, it clears the ref_bit and moves on to the next page:

```
ref_bit = 0;  
curr_ptr++;
```

Case 2: If clock is pointing to a page with `pin_count = 0` and `ref_bit = 0`, it chooses that page as a victim for replacement:

`return curr_ptr;`

`return curr_ptr++;`



In contrast to LRU, clock replacement does not require any work in the common case of accessing a page. The only time work needs to be done is when there is an actual need for page replacement.