Database System

lecture - 10

Memory MGMT : LRU, LRU-K, CAMP

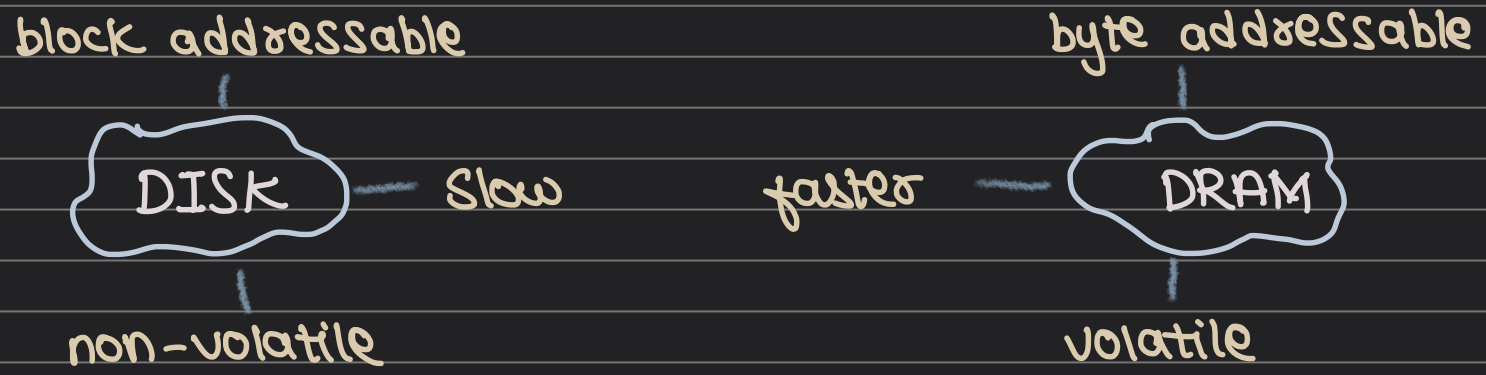
* Memory Management

→ Divide & Conquer

WIN by dividing & conquering the problem — partioning strategies with the data by dividing it.

LOSE by dividing your resources

→ Node Hardware & Disk I/O



All the refrences for the program that's executing on the CPU to go the DRAM & minimize the no. of references going to DISK

Disk read constructs a copy of the disk page in memory if one does not exists.

Array of Bytes

Buffer pool is a data structure with **F frames** & the no. of frames in the buffer pool is significantly smaller than the number of disk pages that compete for those frames.

No. of

Buffer pool pages $<$ Disk pages

Buffer pool manager = a piece of software that executes on the CPU.

& How do you find a page in memory — Buffer pool manager finds it, by maintaining a hash index, (key of the hash index = page ID, value = address of that frame in main memory).

A write updates the copy of a disk page in memory & marks it DIRTY.

Q. When F frames are occupied & there's a request for a new page.

→ It must select a page to evict. (victim page)

→ If the victim page is dirty; then write it to disk & evict.

Q. How to select a victim?

→ Selecting a page that is least frequently used. (LFU)

→ Selecting a page that is least Recently used. (LRU)

(2 different concepts)

→ LRU

The Pseudo-code is not implemented because it runs in a for loop & in step 2 its complexity is O(n) (n: no. of frames)

(Faster implementation)

∴ The solution to the expensive complexity is to implement a doubly linked list of elements. Every time a page is referenced move the page to the end of the list & victim is always at the head of the list.

HEAD ↑ points to the first element

TAIL ↑ points to the last element

HASH TABLE ↑ indexes from page id to its element.

Q How to avoid the overhead of writing a dirty page back to the disk?

→ When the head points to a dirty page, schedule it to be written to disk asynchronously & select the next clean page on the list as a victim.

→ Background threads that write dirty pages back to disk by iterating LRU's doubly linked list.

Here, a page is similar to a key-value pair:

key = page ID ; Value = content of the disk page.

LRU is implemented by cache managers such as memcached to manage their main memory.

The typical architecture is to take a DBMS & augmented with a cache server.

It doesn't consider the time to compute the key value pairs or the cost associated with the key value pairs.

limitation: LRU decides what page to evict from memory based on too little information (time of last reference).

What is missing?
→ Size of key-value pairs.
→ Cost of key-value pairs.
→ Time to compute key-value pairs.

→ Solution:

Partitioning resources (not a winning strategy)

→ Human Tuning: A human being essentially is tasked with identifying grouping of key based on similar size, cost or both, & then partition memory in pools & later assign one or more groups to a pool.
Each pool uses LRU (dividing memory into these pools).

challenges of human tuning:

→ How does the system adminstrator group trillions of key value pairs across a dozen of applications?

→ How do they partition memory?

→ Mapping of 2 or more groups to one partition? Why were they not assigned to the same group to begin with?

→ CAMP: It keeps the resources united by preventing this concept of pooling that the human technique does. It is efficient implementation of GreedyDualSize algorithm. As efficient as LRU & considers size, cost & recency of references.

→ Greedy Dual Size

It initializes a global variable which is assigned to be 0, & then processes each request for key value pairs in turn, so the current request is for key-value pair P.

→ If P is already in main memory (M):
It sets L to be the key value pair that's in main memory that has the lowest priority.

→ Else:
It looks to see if there is enough room in memory for P.
If there is not: then it evicts the key value pair with the smallest priority, & sets the value of L to be this minimum priority key-value pair.
Bring P into memory.

Set the priority of P to be $L + \overbrace{\frac{cost(P)}{size(P)}}^{\text{value of each byte}}$

If the key-value pair is 100 bytes:
$\frac{cost(P)}{size(P)} = \frac{100}{100} = 1$

The more time consuming to compute a key value pair, the higher its cost to size ratio.

Q What is the cost here?

→ The total sum of the execution time of the queries.

→ Cost is based on value, influencers (how influential is this person?).

→ Paid subscriptions, whoever pays more, their assigned cost is higher.

It's similar to LRU, but it's too inefficient to implement.

→ Naive Implementation:

Use a priority Queue implemented as a heap. The values in there is the priority of the key-value pair, And the letters that you see is uniquely identifying key. The victim is at the very top, the one with the lowest priority is the victim.

→ Disadvantages:
→ log n; where n is the no. of key-value pairs.
→ CPU time to maintain the heap becomes significant.

→ CAMP:
Greedy Dual Size is good but it insists to evict the key value pair with the absolute minimum priority value.

(Its combining LRU & GDS)

The design decision of camp, instead evicts the key-value pair whose priority is approximately smallest.

STEP 1:

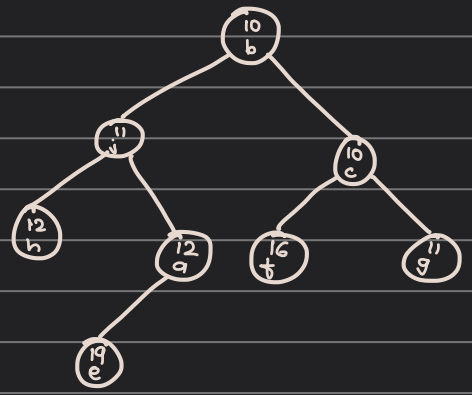
CAMP rounds the priority for every key value pair by rounding its cost to size ratio before adding value L to it.

→ This results in a smaller set of possible cost/size values. Orgnize the key-value pairs with the same cost/size values in an LRU queue.

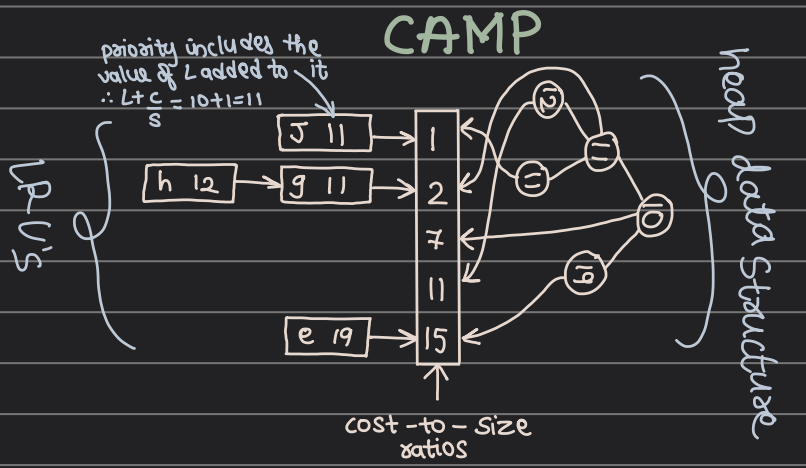
→ Maintain priority across the LRU queue using a heap; Evict from the LRU queue with the lowest priority.

→ GDS VS CAMP

GDS



CAMP



In CAMP the complexity is a function of the unique cost to the size ratios. Whereas with the greedy dual size the complexity is a function of n, where n is the no. of key-value pairs.

☆ One LRU queue for each cost/size ratio. when a key-value is referenced, CAMP adjusts its priority & moves it to the tail of the LRU queue. If key-value pairs within each are stored according to LRU then the key-value pairs are automatically ordered according to their priority.

The first key-value pair in each queue has the lowest priority. It is a candidate victim.

Consider the is a referance for this key=q. we take q & adjust its priority & it becomes 12. we put it at the end of the doubly linked list. we later update the element in the heap & repair the heap structure based on the changes made.

☆ To victimze a key-value pair, look at the root of the heap (because it identifies the minimum value pair). And change the data structure based on the values. with the incoming new key value pair we compute it's cost to size ratio, which will be indexed into that array.

Q. How is the cost/size converted to a real number?
Divide the cost-to-size ratio of a key-value pair by a lower bound estimate on the smallest cost-to-size ration that can ever occur.
Round the resulting integer using higher order bits.

CAMP Rounding:

→ The objective is to assign the key-value pairs with similar cost size values to the same LRU queue to come up with the grouping AND values that have different orders of magnitudes should remain distinct.

→ Preserve the Most Significant Bits (MSB) starting with b, the highest order bit that is not zero. If b is less than p then the value is not rounded.

eg: 101101011 → 101100000
001010011 → 001010000
000001010 → 000001010

A higher precision increases the number of LRU queues.
With precision set to infinity, there are still LRU queues because of similarity in cost & size.

GDS	CAMP
GDS updates its heap every time the priority of a key-value pair is updated.	CAMP updates everytime the priority of a key-value pair is updated.
More memory means more key-value pairs to update.	With larger memory, more items can be stored & there are fewer updates to cache.

→ Evaluation using BG: (comparing LRU & pooled version of LRU)

Use BG to generate a trace of key-value references. Skewed pattern of access with 70% of requests reference 20% of keys. Cost is selected from {1, 100, 100K}. A key-value pair is assigned to a key-value pair, it remains in effect for the entire trace.

Metrics:
Miss rate: no. of misses / total no. of requests.
Cost-Miss ratio: sum the cost for each request. } ideal should be: zero.

→ CAMP is superior to LRU:

It provides a lower cost ratio, because it considers the cost divided by size, whereas, LRU doesn't consider cost at all, its based on the recency of references.

→ CAMP Evolves:

10 traces files each with different set of object references. Once the simulator switches from TF1 & TF2, none of the objects referenced by TF1 are referenced again. Cost-miss ratio & miss rate trends remain the same.

→ Variable Size / Fixed cost.

With variable sized key-value pairs that have identical cost, CAMP renders small key-value pairs less resident. As CAMP considers cost over size, small key

have higher priority & by doing so it can do better job of getting cached hit instead of miss rate.

→ Equi Size/ variable cost.

It is very challenging to construct pools with pooled LRU. CAMP continues to provide a lower cost-miss ratio. The increase is due to greate no. of cost-to-size ratios as a result of the considerably larger set of cost values.

→ When a cached key-value pair should be delete it. Just because it is not reachable, Do NOT depend on LRU or CAMP to evict it.

- CAMP is a memory management technique that considers cost, size, and recency of references to key-value pairs.
- Objective: Minimize cost-miss ratio.
- CAMP does NOT partition resources.
- CAMP adapts to evolving acess patterns.