6.830 Lecture 8
9/29/2014

Last time:

Talked about column-stores. Recap slide.

Where are we:

This time -- Buffer Pool Management.
Next time -- joins.
Then query optimization.

DBMIN

Buffer pool:

cache of recently used pages

convenient "bottleneck" through which references to underlying pages go
useful when checking to see if locks can be acquired or not

Shared between all queries running on the system.

Diagram:

pg id	lock	ptr
1	R/W, TID 2	0xABCD
2	...	0xCDEF

Cache -- <u>so what is the best eviction policy?</u>

<u>LRU?</u> What if a query just does one sequential scan of a file -- then putting it in the cache at all would be pointless. So you should only do LRU if you are going to access a page again, e.g., if it is in the inner loop of a NL join.

For the inner loop of a nested loops join, is LRU always the best policy?
No, if the inner doesn't fit into memory, then LRU is going to evict the record over and over.

E.g., 3 pages of memory, scanning a 4 page file:

pages	A	B	C	read	hit/miss
	1			1	m
	1	2		2	m
	1	2	3	3	m
	~~1~~ 4	2	3	4	m
	~~1~~ 4	~~2~~ 1	3	1	m
				2	m

Always misses?. What would have been a better eviction policy? MRU!

pages	A	B	C	read	hit/miss?
	1			1	m
	1	2		2	m
	1	2	3	3	m
	1	2	~~3~~ 4	4	m
	1	2	4	1	h
	1	2	4	2	h
	1	~~2~~ 3	4	3	m
	1	3	4	4	h
	1	3	4	1	h
	~~1~~ 2	3	4	2	m

Here, MRU hits 2/3 times.

DBMIN tries to do a better job of managing buffer pool by
1) allocating buffer pools on a per-file-instance basis, rather than a single pool for all files
2) using different eviction policies per file

What is a "file instance"?

(Open instance of a file by some access method.)

Each time a file is opened, assign it one of several access patterns, and use that pattern to derive a buffer management policy.

(What does a policy consist of?)

Policy for a file consists of a number of pages to allocate as well as a page replacement policy.

(What are the different types of policies?)

Policies vary according to access patterns for pages.   What are the different access patterns for
pages in a database system?

SS - Straight Sequential (sequential scan)
~~CS - Clustered Sequential (merge join) (skip)~~

LS - Looping sequential (nested loops)
SR - Straight Random (index scan through secondary index)
~~CR - Clustered Random (index NL join with with secondary index on inner, with repeat foreign keys on outer) (skiP~~

SH - Straight Hierarchical (index lookup)
LH - Looping Hierarchical (repeated btree lookups)

So what's the right policy:

SH - 1 page, any access method
~~CS - size of cluster pages, LRU~~
LS - size of file pages, any policy, or MRU plus however many pages you can spare
SR - 1 page, any access method
~~CR - size of cluster pages, LRU~~
SH - 1 page, any access method

LH - top few pages, priority levels, any access method for bottom level

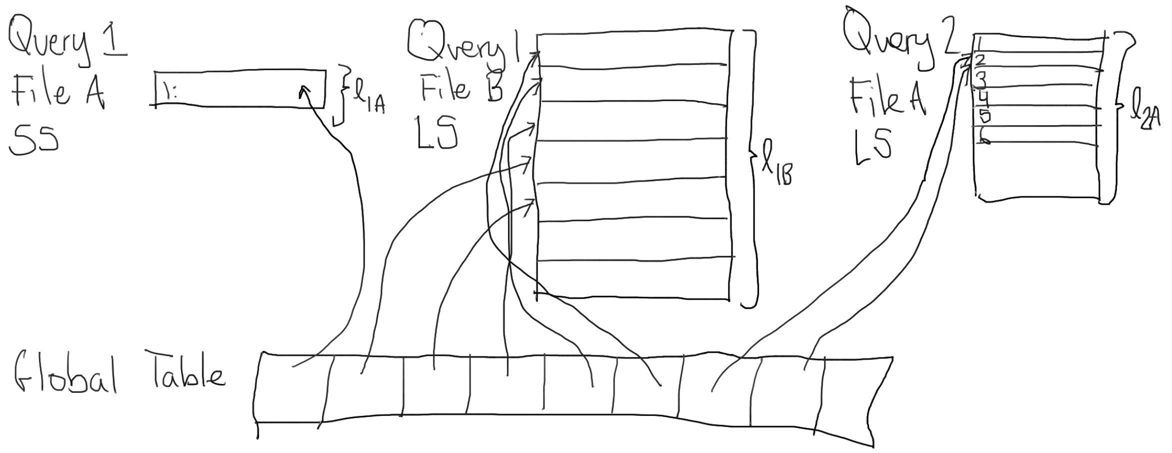
How do you know which policy to use?

(Not said, presumably the query parser/optimizer has a table and can figure this out.)

Break -- demo of pgadmin3

Buffer Manager Implementation.

Diagram:



Buffer pool per file instance, with locality set for that instance, plus "global table" that contains all pages.

Each page is "owned" by a at most one query. Each query has a "locality set" of pages for each file instance it is accessing as a part of its operation, and each locality set is managed according to one of the above policies.

Also store current number of pages associated with a file instance (r) and the maximum number of pages associated with it (l).

How do you determine the maximum number of pages?

Using numbers above.

What happens when the same page is accessed by multiple different queries?

1) Already in buffer pool and owned locally
2) Already in buffer pool, but not owned
   a) If someone else owns, nothing to be done
   b) If no owner, requester becomes owner
3) Not in buffer pool - requester becomes owner, evict something from requester's memory

Lookups just done in global table.

<u>How do you avoid running out memory?</u>

Don't admit queries into the system that will make the total sum of all of the l_ij variables > total system memory.

Metacomments about performance study.

(It's good.) Interesting approach. What did they do?

Collect real access patterns and costs, use them to drive a simulation of the buffer pool.

(Why?) Real system would take a very long time to run, would be hard to control.

<u>How much difference did they conclude this makes?</u>

As much as a factor of 3 for workload with lots of concurrent queries and not much sharing. Seems to be mostly due to admission control. With admission control, simple fifo is about 60% as good as DBMIN.

DBMIN is not used in practice.

What is? (Love hate hints).

What's that? (When an operator finishes with a page, it declares its love or hate for it. Buffer pool preferentially evicts hated pages.)

Not clear why (this would make a nice class project.)

Perhaps love hate hints perform almost as well as DBMIN and are a lot simpler. They don't capture the need for different buffer management policies for different types of files.

(What else might you want the buffer manager to do?)

Prefetch.

(Why does that matter.)

Sequential I/O is a lot faster. If you are doing a scan, you should keep scanning for awhile before servicing some other request, even if the database hasn't yet requested the next page.

Depending on the access method, you may want to selectively enable prefetching.

Generalization: service I/O requests, for reads and writes, sequentially.

E.g., flush pages back in order
Fetch index pages in order when doing a 2ndary index scan

Interaction with the operating system
(What's the relationship between the database buffer manager and the operating system?)

Long history of tension between database designers and OS writers. These days databases are an important enough application that some OSes have support for them.

(What can go wrong?)

- Double buffering -- both OS and database may have a page in memory, wasting RAM.

- Failure to write back -- the OS may not write back a page the database has evicted, which can cause problems if, for example, the database tries to write a log page and then crashes.

- Performance issues -- the OS may perform prefetching, for example, when the database knows it may not need it.

Disk controllers have similar issues (cache, performance tricks.)

(What are some possible solutions?)

- Add hooks to the OS to allow the database to tell it what to do.

- Modify the database to try to avoid caching things the OS is going to cache anyway.

In general, a tension in layered systems that can lead to performance anomalies.