A (parallel) look into
transactional storage managers

Alberto Lerner

NYU - advanced database systems - Fall '11
Mapping Abstractions in Storage

Table:

Row:

ID | John Doe |
---|----------|
...

Bookkeeping

Slots
AN "ACCESS METHOD"

TABLE SCAN
WHO TAKES CARE OF ISOLATION?

ACCESS METHOD

Read

Read access

Look "table"

LOCK MANAGER

Is there a conflict?
Yes: hold
No: take the lock and proceed
Locks vs. Latches

Repeatable Read

Read Locks

Lock Manager

Does Locks

Internal Data Structures Have Latches
Buffer Manager

Controls which pages are in memory

Access method

Pin pages / read rows

Read pages

LRU

FREE

\[ 5 \rightarrow 0 \rightarrow 1 \rightarrow \ldots \]
Durability? Atomicity?

Diagram:
- Access Method
- Log Manager
- 'log'
- Data
- Write
- Commit
- Abort
- Single change
STATE:
- Current block
  knows how to
  "next"

ACCESS METHOD

LOOK MANAGER

LOG MANAGER

BUFFER MANAGER

STATE:
- Used frames
  knows which
  frame can be
  used

STATE:
- LSN/log records
  knows how
  to undo/redo

Hillerstein, Stonebreaker, and Hamilton
"Architecture of a Database System"
What happens to this scheme if we're running on parallel hardware?

Eg. NUMA (A.K.A. your vanilla workstation)
GOAL

→ TAKE ADVANTAGE OF THE PARALLEL HARDWARE FOR RUNNING MORE QUERIES FASTER.

LET'S ASSUME OUR WORKLOAD IS NOT DISK BOUND SO IF THESE SYSTEMS CAN'T UPDATE STATE FAST ENOUGH THEY BECOME BOTTLENECKS.
Non-goals
(fun but we have enough on our plate for today)

→ To execute just one — but heavy — query faster

→ We keep ACID

If we change the transactional guarantees, a whole new design space opens up

→ Speaking of design, we assume we need these four sub-systems

In other words, let’s see how each behaves in a parallel setting

→ Parallel, not partitioned nor distributed
  (fun! but I repeat myself)
What are the problems?

- State that needs to be updated on a per-row basis or otherwise frequently.

- The machine can potentially do those in parallel.

- The "managers" have more or less problems taking advantage of it.
Buffer Manager

Goals

Is page already in memory?

Pick a victim

State:

- Frames in use
- Recency
• CONCURRENT ACCESS TO THE HASH TABLE

• LOCK STRIPING IS AN EASY AND EFFICIENT SOLUTION

so far, so good
For evictions, we'd like to maintain recency info (e.g., LRU). What happens to the recency list(s) at every accessed page?

We actually maintain a "hot" and a "less hot" list.
- Imagine the nodes (frames in use) are still connected through a list.

- But now, eviction is done by traversing the list "clockwise".

- Each node has a counter decreased when the "clock arm" moves, and increased (or set) when the page is accessed.

- A node with count zero can be evicted.
WHAT CHANGED?

- THE REGENCY LIST ITSELF DOES NOT GET UPDATED AT EVERY PAGE HIT

- WE CAN PLAY WITH THE INITIAL COUNTER VALUE OR THE INCREMENT

- WE DON'T GET EXACT LRU BUT AN APPROXIMATION OF IT
BUFFER MANAGER REFERENCES

- Corbato, "A Paging Experiment with the MULTICS System", MIT Project MAC Report MAC-M-384, May, 68
- Johnson and Shasha, "2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm", VLDB 94
- Jiang et al., "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement", USENIX 05
- Yui et al., "NB-GLOCK: A Non-Blocking Buffer Management Based on the Generalized CLOCK", ICDE 10
LOG MANAGER

GOAL

One way to look at it is giving the buffer manager the freedom to flush a page whenever it wants.

- If a TX is rolled back for which a page has been written, the log manager can undo the changes.
- If a TX commits for which there are page changes, the log manager can redo the changes in case of a crash.

All it requires is that the TX's changes (the log) be flushed before the page changes.
WRITE AHEAD LOG

IF THE BUFFER MANAGER NEEDS TO FLUSH A PAGE, THE LOG MANAGER MUST FIRST FLUSH UNTIL THAT LSN.

LOG

DATA
WHAT ARE THE CONTENTION POINTS HERE?

(A) THERE'S A LATCH ON THE PAGE TO UPDATE THE LSN, EVEN IF, SAY, TWO TX'S ARE TOUCHING TWO DISTINCT ROWS

(B) APPENDING TO THE LOG BUFFER IS A SERIALIZATION POINT

(C) THE COMMITTING TX HOLDS LOCKS (NOT LATCHES) AND WOULD CONTINUE TO DO SO THROUGHOUT THE LOG FLUSH
(c) FIRST

→ THE PROBLEM GOES AWAY IF THE TX CAN COMMIT "ASYNCHRONOUSLY"

- RELEASE THE LOCKS BEFORE COMMITTING, BUT DON'T RETURN UNTIL DURABLE

("Partial Strictness in Lock Release")
(B) CONTENTION WRITING TO THE LOG BUFFER

-  **Decouple producing and consuming ends of the queue**

-  **Note that group commit occurs naturally here**

-  **Possibly combine mods of tx's waiting on the producer latch**
(a) LSN per page

- In Aries-based systems, the unit of recovery is a page - even if concurrency among objects within a page is conceivable.

- Recent attempt at segment-based recovery:
  - A segment is just a range of bytes.
  - It may span several disk pages.
  - There may be many segments in a page.

Can we pack entirely different structures without worrying about the page concept?
LOG MANAGER REFERENCES


- Sears and Brewer, "Segment Based Recovery: Write Ahead Logging Revisited", VLDB 09
LOCK MANAGER

GOALS:

→ SCHEDULES TRANSACTIONS BY WAY OF LOCKS AND INTENT LOCKS

→ DETECTS TIMEOUTS AND DEADLOCKS

STATE:

→ CURRENT LOCKS AND LOCKS FOR A TRANSACTION
Acquiring a Lock

1. Is the higher-level intent lock taken?
2. Is the actual lock taken?

If so, is the lock mode requested compatible with the current mode?

Example:

Lock S on a row is on the page

\[
\text{Row} / \begin{cases} \text{S} \\ \text{S} \end{cases} \rightarrow \begin{cases} \text{S} \\ \text{S} \end{cases} \rightarrow \begin{cases} \text{S} \\ \text{S} \end{cases} \rightarrow \begin{cases} \text{S} \\ \text{S} \end{cases}
\]
The lock manager also keeps track of which lock belongs to which transaction.
RELEASING A TX'S LOCK

1. FOR EACH LOCK IN THE TRANSACTION'S CHAIN
   1. UNCHAIN IT
   1. IF LOCKS WAITING, DECIDE WHO IS ACQUIRING NEXT
   1. COMPUTE NEW LOCK MODE

OF COURSE, THE LOCK TABLE INTEGRITY HAS TO BE MAINTAINED THROUGHOUT THE PROCESS (HOW?)

NOTE THAT WE MAY BE DOING ALL THAT JUST TO ACCESS A ROW IN A PAGE THAT IS ALREADY PINNED (WHAT ARE THE RELATIVE COSTS?)
WHAT IF A LOCK CAN'T BE ACQUIRED IMMEDIATELY?

1. Block the requesting thread (TX) until the lock is available.
   (This may mean a context switch and it depends on what the process model is.)

2. "Spin" in the hope that the lock would be available soon.

The best option depends also on the current load of the machine.
Spinning while the lock is taken can be efficiently done as long as we don't prevent the lock holder from running.

Blocking prevents wasting cycles.

From Johnson et al. '10.
Lock Manager References

- Johnson et al., "Decoupling Contention Management from Scheduling", ASPLOS '10

- Gray and Reuter, "Transaction Processing: Concepts and Techniques", MK '93

- Bernstein, Hadzilacos, and Goodman, "Concurrency Control and Recovery in Database Systems", AW '87
Access Methods

Goals

- To encode a given page/row traversal logic

Examples:

  - Table Scan
  - Index Scan

- Query Operators are defined using the access methods
Let's look at an index traversal method.

To guarantee that the tree won't structurally change during traversal, we latch higher levels when descending.

An interesting algorithm to delay splits and improve consistency is due to Lehman and Yao '81 (with some refinements by Jaluta et al. '05).
Sources of Problems

1. We can fetch a page and then have to wait for space in the buffer pool.
Sources of Problems

2. Threads are going to compete for latches and, if pages change, invalidate caches.
What if we could avoid latching altogether?

One idea that has been explored recently is to "confine" the scope that each thread can manipulate.

A query is handled by the thread that touches the relevant data partition.
It is not without its complications

- What about queries that touch multiple partitions?

- What about load balancing of partitions?
A second front

- What about search data structures that are easier to parallelize?

- An example: skip lists

  Pro: No balancing

  Con (?): How to pack the structure?
ACCESS METHODS REFERENCES

- Lehman and Yao, "Efficient Locking for Concurrent Operations on B Trees", ACM TODS 6(4), 84

- Talota et al., "B-Tree Concurrency Control and Recovery in Page-Server Database Systems", ACM TODS 31(1), 06

- Sewall et al., "PALM: Parallel Architecture-Friendly Latch Free Modifications to B+ Trees on Many-Core Processors", VLDB '11

- Pandis et al., "PLF: Page Latch-Free Shared Everything OLTP", VLDB '11

- William Pugh, "Concurrent Maintenance of Skip Lists", TR-2222.1, U. of Maryland, 89
Transational storage systems have been built since... there were storage.

Gray/Reuter 193 is still an excellent source.

But some important changes since:

- Workloads are more diversified.
  
  (Back in the 1980s, what was the largest workload? And now?)

- Parallel machines and hierarchies of memory are ubiquitous now.
So what are database system people doing?

- Exploring recovery techniques that don't require page boundaries
- Experimenting providing weaker transactional guarantees
- Trying more data-centric execution style as opposed to task-centric one
- Considering more architecture-friendly data structures

Some, driven by specific need, decide to part with the "traditional" architecture altogether