Transaction Processing: Recovery

CPS 216
Advanced Database Systems

---

Announcements (April 28)

❖ Homework #4 due today
- Sample solution will be emailed to you by tomorrow morning

❖ Project demo period: April 28 – May 1
- Remember to email me to sign up for a 30-minute slot

❖ Final exam on Monday, May 2, 2-5pm
- 3 hours—no time pressure!
- Open book, open notes
- Comprehensive, but with emphasis on the second half of the course and materials exercised in homework

❖ Solution to sample final available

Review

❖ ACID
- Atomicity
- Consistency
- Isolation ⟶ Concurrency control
- Durability ⟶ Recovery

Execution model

CPU   Memory X $Y...$ Disk X $Y...$
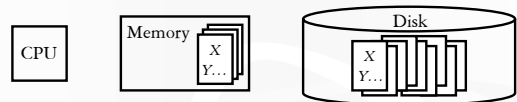
❖ Before it can be operated upon, disk-resident data must first be brought into memory
- input(X): copy the disk block containing object X to memory
- v = read(X): read the value of X into a local variable v
  - Execute input(X) first if necessary ⟶ Issued by transactions
- write(X, v): write value v to X in memory
  - Execute input(X) first if necessary ⟶ Issued by DBMS
- output(X): write the memory block containing X to disk

Failures

❖ System crashes in the middle of a transaction T; partial effects of T were written to disk
- How do we undo T (atomicity)?

❖ System crashes right after a transaction T commits; not all effects of T were written to disk
- How do we complete T (durability)?

❖ Media fails; data on disk corrupted
- How do we reconstruct the database (durability)?

Naïve approach

❖ Force: When a transaction commits, all writes of this transaction must be reflected on disk
- Without force, if system crashes right after T commits, effects of T will be lost
- ☞ Problem: Lots of random writes hurt performance

❖ No steal: Writes of a transaction can only be flushed to disk at commit time
- With steal, if system crashes before T commits but after some writes of T have been flushed to disk, there is no way to undo these writes
- ☞ Problem: Holding on to all dirty blocks requires lots of memory

Logging

❖ Log
  - Sequence of log records, recording all changes made to the database
  - Written to stable storage (e.g., disk) during normal operation
  - Used in recovery

❖ Hey, one change turns into two—bad for performance?
  - But writes are sequential (append to the end of log)
  - Can use dedicated disk(s) to improve performance

Undo/redo logging rules

❖ Record values before and after each modification:
  $\langle T_i, X, old\_value\_of\_X, new\_value\_of\_X \rangle$
❖ A transaction T_i is committed when its commit log record $\langle T_i, \text{commit} \rangle$ is written to disk
❖ Write-ahead logging (WAL): Before X is modified on disk, the log record pertaining to X must be flushed
  - Without WAL, system might crash after X is modified on disk but before its log record is written to disk—no way to undo
❖ No force: A transaction can commit even if its modified memory blocks have not be written to disk (since redo information is logged)
❖ Steal: Modified memory blocks can be flushed to disk anytime (since undo information is logged)

Undo/redo logging example

T_1 (balance transfer of \$100 from A to B)

read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);
commit;

Memory

$A = \cancel{800}\,700$
$B = \cancel{400}\,500$

Disk

$A = \cancel{800}\,700$
$B = \cancel{400}\,500$

Log

$<T_1, \text{start}>$
$<T_1, A, {}_{800, 700}>$
$<T_1, B, {}_{400, 500}>$
$<T_1, \text{commit}>$

Steal: can flush before commit

No force: can flush after commit

No restriction on when memory blocks can/should be flushed

Checkpointing

❖ Naïve approach:
  - Stop accepting new transactions (lame!)
  - Finish all active transactions
  - Take a database dump
  - Now safe to truncate the log

❖ Fuzzy checkpointing
  - Determine S, the set of currently active transactions, and log $\langle \text{begin-checkpoint } S \rangle$
  - Flush all modified memory blocks at your leisure
  - Log $\langle \text{end-checkpoint } begin\text{-}checkpoint\_location \rangle$
  - Between begin and end, continue processing old and new transactions

Recovery: analysis and redo phase

❖ Need to determine U, the set of active transactions at time of crash
❖ Scan log backward to find the last end-checkpoint record and follow the pointer to find the corresponding $\langle \text{start-checkpoint } S \rangle$
❖ Initially, let U be S
❖ Scan forward from that start-checkpoint to end of the log
  - For a log record $\langle T, \text{start} \rangle$, add T to U
  - For a log record $\langle T, \text{commit} \mid \text{abort} \rangle$, remove T from U
  - For a log record $\langle T, X, old, new \rangle$, issue write($X$, new)
  ☞ Basically repeats history!

Recovery: undo phase

❖ Scan log backward
  - Undo the effects of transactions in U
  - That is, for each log record $\langle T, X, old, new \rangle$ where T is in U, issue write(X, old), and log this operation too (part of the repeating-history paradigm)
  - Log $\langle T, \text{abort} \rangle$ when all effects of T have been undone

☞ An optimization
  - Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo

Physical vs. logical logging

- ❖ Physical logging (what we have assumed so far)
  - ▪ Log before and after images of data
- ❖ Logical logging
  - ▪ Log operations (e.g., insert a row into a table)
  - ▪ Smaller log records
    - • An insertion could cause rearrangement of things on disk
    - • Or trigger hundreds of other events
  - ▪ Sometimes necessary
    - • Assume row-level rather than page(block)-level locking
    - • Data might have moved to another block at time of undo!
  - ▪ Much harder to make redo/undo idempotent
  - ☞ See solution offered by ARIES

ARIES

"ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," by Mohan et al. *TODS* 1992

- ❖ Same basic ideas: steal, no force, WAL
- ❖ Three phases: analysis, redo, undo
  - ▪ Repeats history (redo even incomplete transactions)
- ❖ Better than our simple algorithm
  - ▪ CLR (Compensation Log Record) for transaction aborts
  - ▪ Redo/undo on an object is only performed when necessary → idempotency requirement lifted → logical logging supported
    - • Each disk block records the LSN (log sequence number) of the last change
  - ▪ Can take advantage of a partial checkpoint
    - • Recovery can start from any start-checkpoint, not necessarily one that corresponds to an end-checkpoint

Summary

- ❖ Concurrency control
  - ▪ Serial schedule: no interleaving
  - ▪ Conflict-serializable schedule: no cycles in the precedence graph; equivalent to a serial schedule
  - ▪ 2PL: guarantees a conflict-serializable schedule
  - ▪ Strict 2PL: also guarantees recoverability
- ❖ Recovery: undo/redo logging with fuzzy checkpointing
  - ▪ Normal operation: write-ahead logging, no force, steal
  - ▪ Recovery: first redo (forward), and then undo (backward)

Review: XML

- ❖ Data model: tree or graph (with ID/IDREF)
  - ▪ DTD (schema) is optional
- ❖ Query languages: XPath (building blocks for other languages: path expressions), XQuery (SQL-like), XSLT (structural recursion)
- ❖ XML-relational mapping: schema-oblivious (nodes/edges; intervals; label-paths; Dewey order) vs. schema-aware
- ❖ XML query processing: navigational (equality joins) vs. structural (containment joins)
  - → Path expression processing boils down to joins!
- ❖ XML indexing: nodes/edges; intervals; paths; sequences; structural

Review: query optimization or "goodification"?

- ❖ Heuristics: push selections down; smaller joins first
  - → Reduce the size of intermediate results
- ❖ Cost-based
  - ▪ Query rewrite: merge blocks to get a bigger search space
  - ▪ Cost estimation: use statistics (e.g., histograms)
  - ▪ Search algorithm: dynamic programming (+ interesting orders), randomized search, genetic programming, etc.

Review: transaction processing

- ❖ ACID properties
- ❖ Concurrency control
  - ▪ Locking-based: strict 2PL; handling deadlocks; multiple-granularity locking; index and predicate locking
  - ▪ Validation-based, timestamp-based, multi-version
  - → Trade-off: blocking versus aborts and restarts
- ❖ Recovery
  - ▪ Steal: requires undo logging
  - ▪ No force: requires redo logging
  - ▪ WAL (log holds the truth)