5. Database System Recovery

CSEP 545 Transaction Processing for E-Commerce

Philip A. Bernstein

Outline

1. Introduction
2. Recovery Manager
3. Two Non-Logging Algorithms
4. Log-based Recovery
5. Media Failure

1. Introduction

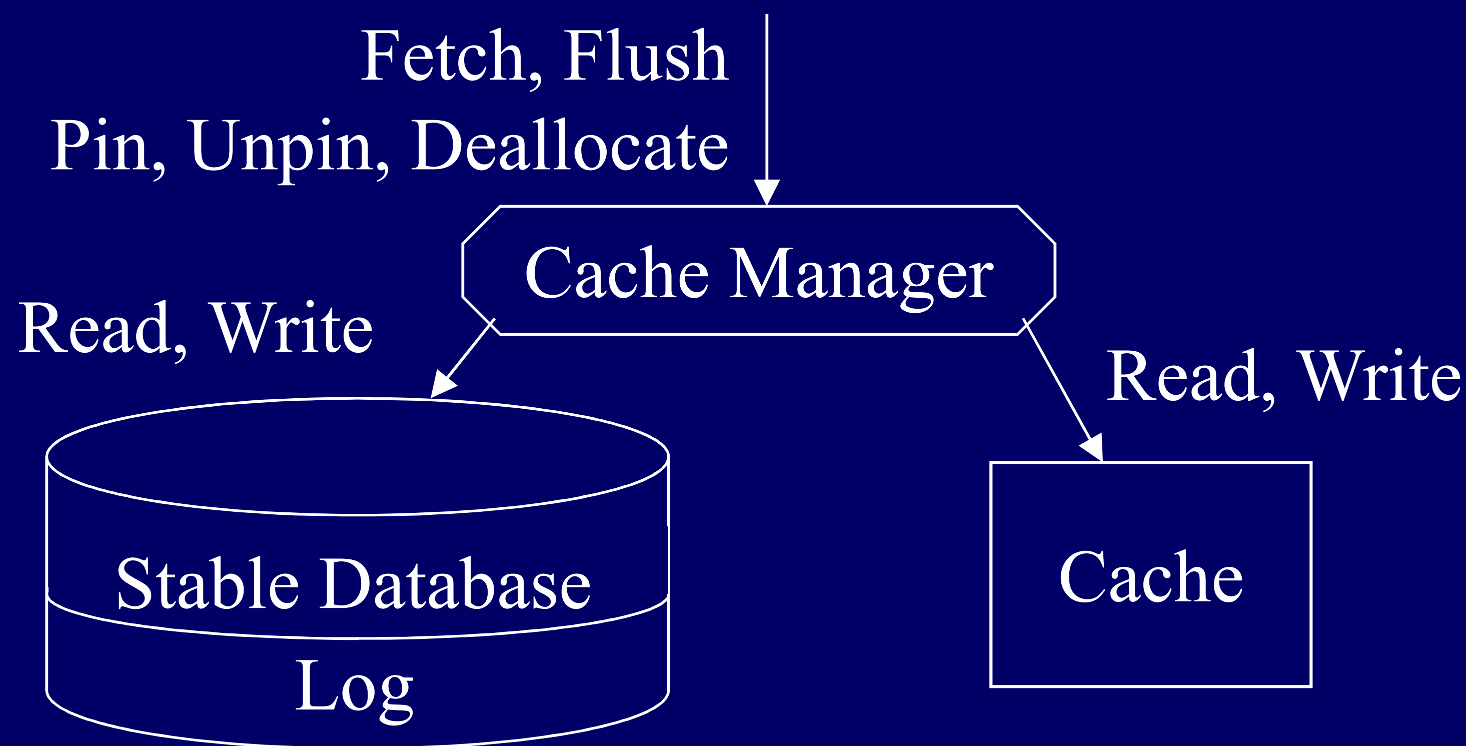
- A database may become inconsistent because of a
  - transaction failure (abort)
  - database system failure (possibly caused by OS crash)
  - media crash (disk-resident data is corrupted)
- The recovery system ensures the database contains exactly those updates produced by committed transactions
  - I.e. atomicity and durability, despite failures

Assumptions

- Two-phase locking, holding write locks until <u>after</u> a transaction commits. This implies
  - recoverability
  - no cascading aborts
  - strictness (never overwrite uncommitted data)
- Page-level everything (for now)
  - page-granularity locks
  - database is a set of pages
  - a transaction's read or write operation operates on an entire page
  - we'll look at record granularity later

Storage Model

- Stable database - survives system failures
- Cache (volatile) - contains copies of some pages, which are lost by a system failure



Stable Storage

- Write(P) overwrites the entire contents of P on the disk
- If Write is unsuccessful, the error might be detected on the next read ...
  – e.g. page checksum error => page is corrupted
- … or maybe not
  – Write correctly wrote to the wrong location
- Write is the only operation that’s atomic with respect to failures and whose successful execution can be determined by recovery procedures.

The Cache

- Cache is divided into page-sized slots.
- Dirty bit tells if the page was updated since it was last written to disk.
- Pin count tells number of pin ops without unpins

Page	Dirty Bit	Cache Address	Pin Count
P_2	1	91976	1
P_{47}	0	812	2
P_{21}	1	10101	0

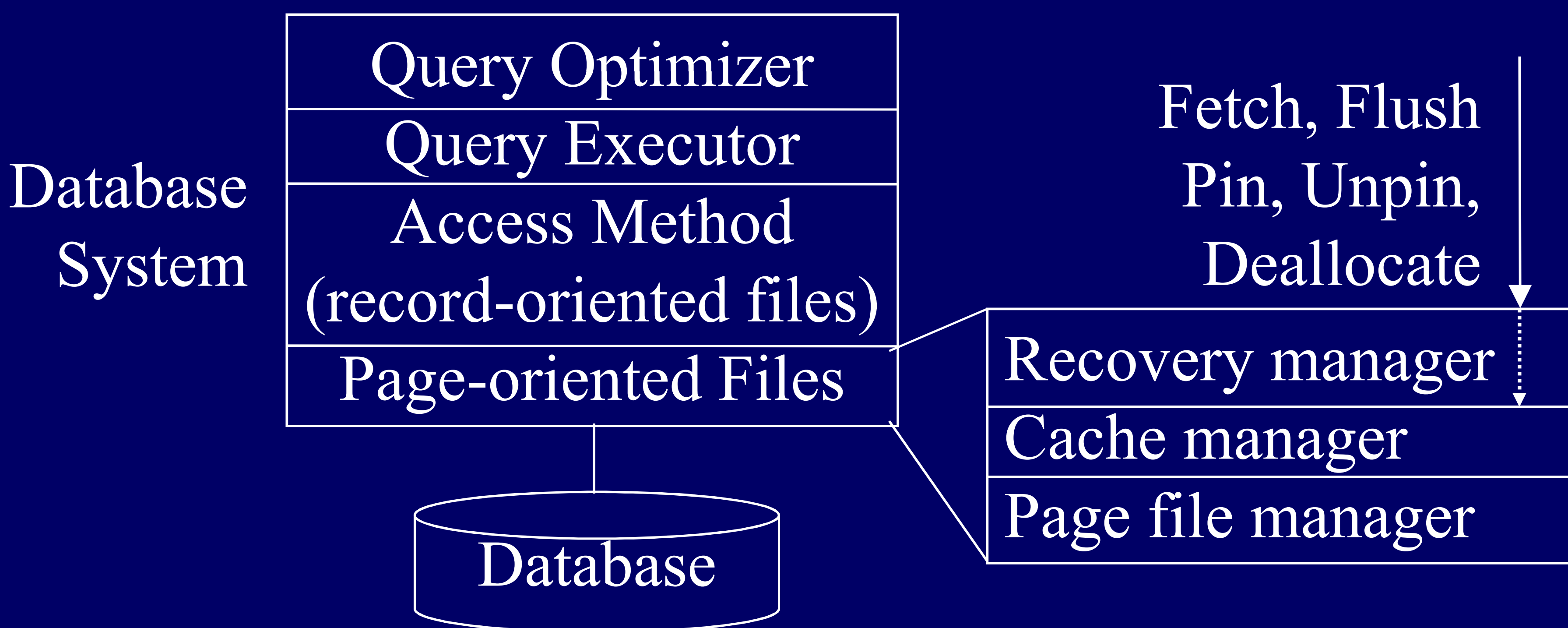
- Fetch(P) - read P into a cache slot. Return slot address.
- Flush(P) - If P's slot is dirty and unpinned, then write it to disk (i.e. return after the disk acks)

Cache (cont'd)

- Pin(P) - make P's slot non-flushable & non-replaceable.
  - Non-flushable because P's content may be inconsistent
  - Non-replaceable because someone has a pointer into P or is accessing P's content.
- Unpin(P) - releases it.
- Deallocate(P) - allow P's slot to be reused (even if dirty)

Big Picture

- Record manager is the main user of the cache manager.
- It calls Fetch(P) and Pin(P) to ensure the page is in main memory, non-flushable, and non-replaceable.



Latches

- A latch is a short-term lock that gives its owner access to a page.
- A read latch allows the owner to read the content.
- A write latch allows the owner to modify the content.
- The latch is usually a bit in a control structure, not an entry in the lock manager. It can be set and released much faster than a lock.
- There's no deadlock detection for latches.

The Log

- A sequential file of records describing updates:
  - address of updated page
  - id of transaction that did the update
  - before-image and after-image of the page
- Whenever you update the cache, also update the log
- Log records for Commit(T_i) and Abort(T_i)
- Some older systems separated before-images and after-images into separate log files.
- If op_i conflicts with and executes before op_k, then op_i's log record must precede op_k's log record
  - recovery will replay operations in log-record-order

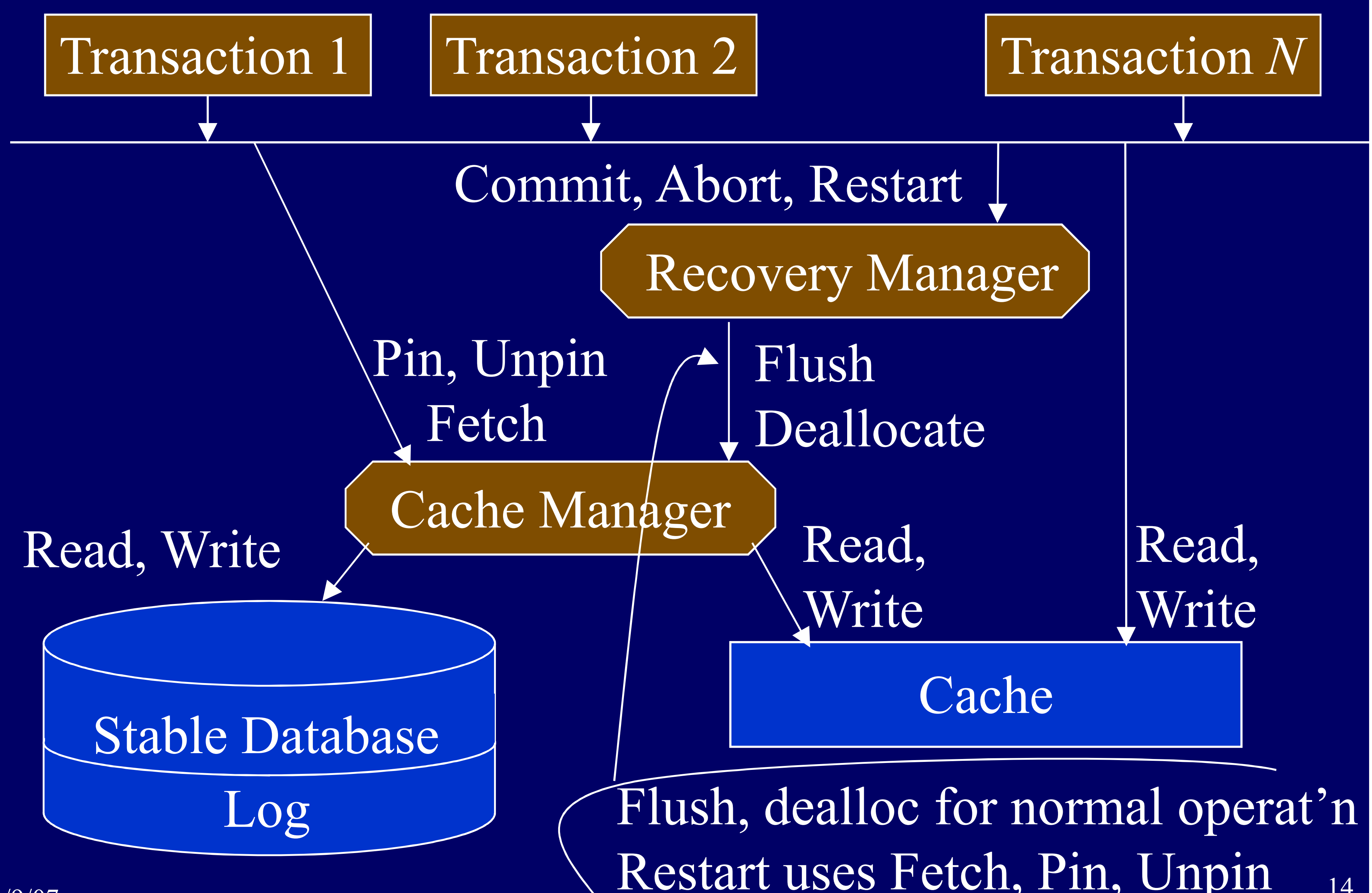
The Log (cont'd)

- To update records on a page:
  - Fetch(P) read P into cache
  - Pin(P) ensure P isn't flushed
  - write lock (P) for two-phase locking
  - write latch (P) get exclusive access to P
  - update P update P in cache
  - log the update to P append it to the log
  - unlatch (P) release exclusive access
  - Unpin(P) allow P to be flushed

2. Recovery Manager

- Processes Commit, Abort and Restart
- Commit(T)
  - Write T's updated pages to stable storage <u>atomically, even if the system crashes.</u>
- Abort(T)
  - Undo the effects of T's writes
- Restart = recover from system failure
  - Abort all transactions that were not committed at the time of the previous failure
  - Fix stable storage so it includes all committed writes and no uncommitted ones (so it can be read by new txns)

Recovery Manager Model



Implementing Abort(T)

- Suppose T wrote page P.
- If P was not transferred to stable storage, then deallocate its cache slot
- If it was transferred, then P's before-image must be in stable storage (else you couldn't undo after a system failure)
- Undo Rule - Do not flush an uncommitted update of P until P's before-image is stable. (Ensures undo is possible.)
  - Write-Ahead Log Protocol - Do not … until P's before-image is in the log

Avoiding Undo

- Avoid the problem implied by the Undo Rule by never flushing uncommitted updates.
  - Avoids stable logging of before-images
  - Don’t need to undo updates after a system failure
- A recovery algorithm requires undo if an update of an uncommitted transaction can be flushed.
  - Usually called a steal algorithm, because it allows a dirty cache page to be “stolen.”

Implementing Commit(T)

- Commit must be atomic. So it must be implemented by a disk write.
- Suppose T wrote P, T committed, and then the system fails. P must be in stable storage.
- Redo rule - Don't commit a transaction until the after-images of all pages it wrote are in stable storage (in the database or log). (Ensures redo is possible.)
  - Often called the Force-At-Commit rule

Avoiding Redo

- To avoid redo, flush all of T's updates to the stable database before it commits. (They must be in stable storage.)
  - Usually called a Force algorithm, because updates are forced to disk before commit.
  - It's easy, because you don't need stable bookkeeping of after-images
  - But it's inefficient for hot pages. (Consider TPC-A/B.)
- Conversely, a recovery algorithm requires redo if a transaction may commit before all of its updates are in the stable database.

Avoiding Undo <u>and</u> Redo?

- To avoid both undo and redo
  - never flush uncommitted updates (to avoid undo), and
  - flush all of T's updates to the stable <u>database</u> before it commits (to avoid redo).
- Thus, it requires installing all of a transaction's updates into the stable database in one write to disk
- It <u>can</u> be done, but it isn't efficient for short transactions and record-level updates.
  - Use shadow paging.

Implementing Restart

- To recover from a system failure
  - Abort transactions that were active at the failure
  - For every committed transaction, redo updates that are in the log but not the stable database
  - Resume normal processing of transactions
- Idempotent operation - many executions of the operation have the same effect as one execution
- Restart must be idempotent. If it's interrupted by a failure, then it re-executes from the beginning.
- Restart contributes to unavailability. So make it fast!

3. Log-based Recovery

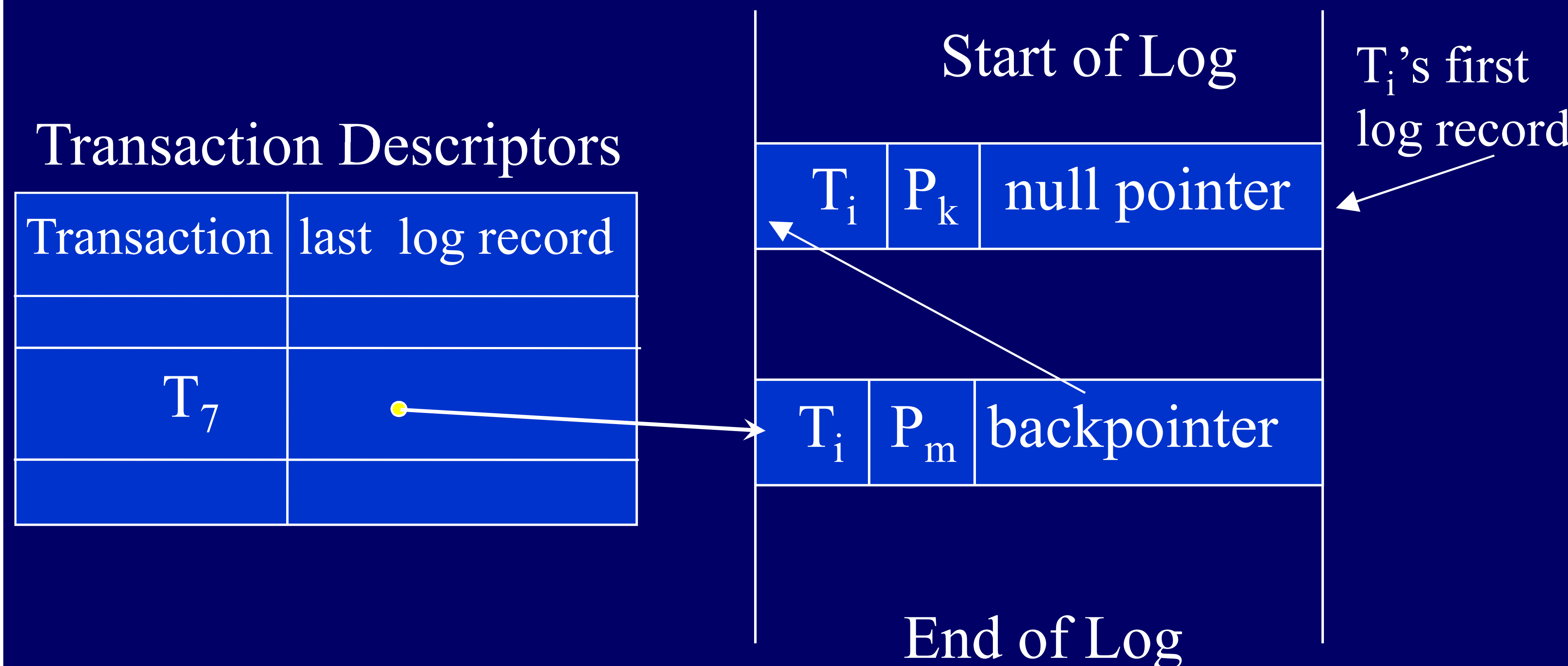
- Logging is the most popular mechanism for implementing recovery algorithms.
- The recovery manager implements
  - Commit - by writing a <u>commit record</u> to the log and flushing the log (satisfies the Redo Rule)
  - Abort - by using the transaction's log records to restore before-images
  - Restart - by scanning the log and undoing and redoing operations as necessary
- The algorithms are fast since they use sequential log I/O in place of random database I/O. They greatly affect TP and Restart performance.

Implementing Commit

- Every commit requires a log flush.
- If you can do K log flushes per second, then K is your maximum transaction throughput
- Group Commit Optimization - when processing commit, if the last log page isn't full, delay the flush to give it time to fill
- If there are multiple data managers on a system, then each data mgr must flush its log to commit
  - If each data mgr isn't using its log's update bandwidth, then a shared log saves log flushes
  - A good idea, but rarely supported commercially

Implementing Abort

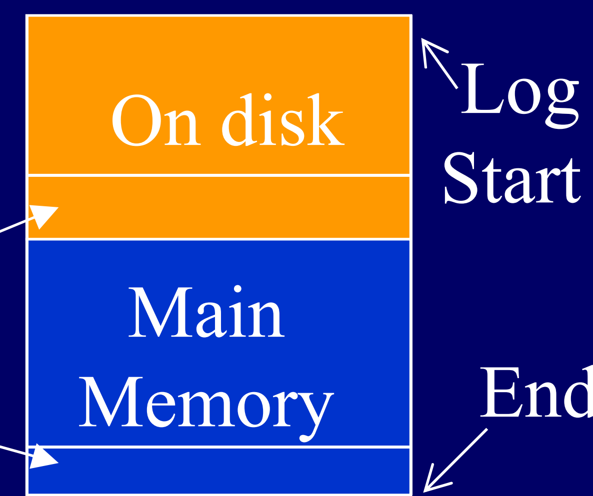
- To implement Abort(T), scan T's log records and install before images.
- To speed up Abort, back-chain each transaction's update records.



Satisfying the Undo Rule

- To implement the Write-Ahead Log Protocol, tag each cache slot with the log sequence number (LSN) of the last update record to that slot's page.

Page	Dirty Bit	Cache Address	Pin Count	LSN
P_{47}	1	812	2	
P_{21}	1	10101	0	



- Cache manager won't flush a page P until P's last updated record, pointed to by LSN, is on disk.
- P's last log record is usually stable before Flush(P), so this rarely costs an extra flush
- LSN must be updated while latch is held on P's slot

Implementing Restart (rev 1)

- Assume undo and redo are required
- Scan the log backwards, starting at the end.
  - How do you find the end?
- Construct a commit list and recovered-page-list during the scan (assuming page level logging)
- Commit(T) record => add T to commit list
- Update record for P by T
  - if P is not in the recovered-page-list then
    - add P to the recovered-page-list
    - if T is in the commit list, then redo the update, else undo the update

Checkpoints

- Problem - Prevent Restart from scanning back to the start of the log
- A <u>checkpoint</u> is a procedure to limit the amount of work for Restart
- Cache-consistent checkpointing
  - Stop accepting new update, commit, and abort operations
  - Make list of [active transaction, pointer to last log record]
  - Flush all dirty pages
  - Append a checkpoint record to log; include the list
  - Resume normal processing
- Database and log are now mutually consistent

Restart Algorithm (rev 2)

- No need to redo records before last checkpoint, so
  - Starting with the last checkpoint, scan forward in the log.
  - Redo all update records. Process all aborts.
    Maintain list of active transactions (initialized to content of checkpoint record).
  - After you're done scanning, abort all active transactions
- Restart time is proportional to the amount of log after the last checkpoint.
- Reduce restart time by checkpointing frequently.
- Thus, checkpointing must be cheap.

Fuzzy Checkpointing

- Make checkpoints cheap by avoiding synchronized flushing of dirty cache at checkpoint time.
  - Stop accepting new update, commit, and abort operations
  - Make a list of all dirty pages in cache
  - Make list of [active transaction, pointer to last log record]
  - Append a checkpoint record to log; include the list
  - Resume normal processing
  - Initiate low priority flush of all dirty pages
- Don't checkpoint again until all of the last checkpoint's dirty pages are flushed
- Restart begins at second-to-last (penultimate) checkpoint.
- Checkpoint frequency depends on disk bandwidth

Operation Logging

- Record locking requires (at least) record logging.
  - Suppose records x and y are on page P
  - $w_1[x]$ $w_2[y]$ $abort_1$ $commit_2$ (not strict w.r.t. pages)
- Record logging requires Restart to read a page before updating it. This reduces log size.
- Further reduce log size by logging <u>description</u> of an update, not the entire before/after image of record.
  - Only log after-image of an insertion
  - Only log fields being updated
- Now Restart can't blindly redo.
  - E.g., it must not insert a record twice

LSN-based logging

- Each database page P's header has the LSN of the last log record whose operation updated P.
- Restart compares log record and page LSN before redoing the log record's update U.
  - Redo the update only if LSN(P) < LSN(U)
- Undo is a problem. If U's transaction aborts and you undo U, what LSN to put on the page?
  - Suppose T_1 and T_2 update records x and y on P
  - $w_1[x]$ $w_2[y]$ c_2 a_1 (what LSN does a_1 put on P?)
  - not LSN before $w_1[x]$ (which says $w_2[y]$ didn't run)
  - not $w_2[y]$ (which says $w_1[x]$ wasn't aborted)

LSN-based logging (cont'd)

- $w_1[x]$ $w_2[y]$ c_2 a_1 (what LSN does a_1 put on P?)
- Why not use a_1's LSN?
  - must latch all of T_1's updated pages before logging a_1
  - else, some $w_3[z]$ on P′ could be logged after a_1 but be executed before a_1, leaving a_1's LSN on P′ instead of $w_3[z]$'s.

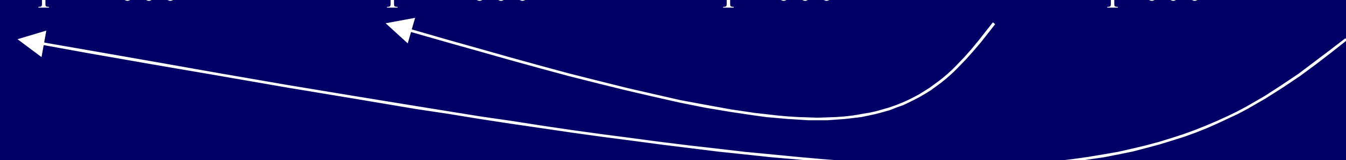
Logging Undo's

- Log the undo(U) operation, and use its LSN on P
  - CLR = Compensation Log Record = a logged undo
  - Do this for all undo's (during normal abort or recovery)
- This preserves the invariant that the LSN on each page P exactly describes P's state relative to the log.
  - P contains all updates to P up to and including the LSN on P, and no updates with larger LSN.
- So every aborted transaction's log is a palindrome of update records and undo records.
- Restart processes Commit and Abort the same way
  - It redoes the transaction's log records.
  - It only aborts active transactions after the forward scan

Logging Undo's (cont'd)

- Tricky issues
  - Multi-page updates (it's best to avoid them)
  - Restart grows the log by logging undos.
    Each time it crashes, it has more log to process
- Optimization - CLR points to the transaction's log record preceding the corresponding "do".
  - Splices out undone work
  - Avoids undoing undone work during abort
  - Avoids growing the log due to aborts during Restart

DoA_1 … DoB_1 … DoC_1 … $UndoC_1$ … $UndoB_1$ …



Restart Algorithm (rev 3)

- Starting with the penultimate checkpoint, scan forward in the log.
  - Maintain list of active transactions (initialized to content of checkpoint record).
  - Redo an update record U for page P only if LSN(P) < LSN(U).
  - After you're done scanning, abort all active transactions. Log undos while aborting. Log an abort record when you're done aborting.
- This style of record logging, logging undo's, and replaying history during restart was popularized in the ARIES algorithm by Mohan et al at IBM.

Analysis Pass

- Log <u>flush</u> record after a flush occurs (to avoid redo)
- To improve redo efficiency, pre-analyze the log
  - Requires accessing only the log, not the database
- Build a Dirty Page Table that contains list of dirty pages and, for each page, the oldestLSN that must be redone
  - Flush(P) says to delete P from Dirty Page Table
  - Write(P) adds P to Dirty Page Table, if it isn't there
  - Include Dirty Page Table in checkpoint records
  - Start at last checkpt record, scan forward building the table
- Also build list of active txns with lastLSN

Analysis Pass (cont’d)

- Start redo at oldest oldestLSN in Dirty Page Table
  - Then scan forward in the log, as usual
  - Only redo records that might need it,
    that is, those where LSN(redo record) ≥ oldestLSN,
    hence there’s no later flush record
  - Also use Dirty Page Table to guide page prefetching
    - Prefetch pages in oldestLSN order in Dirty Page Table

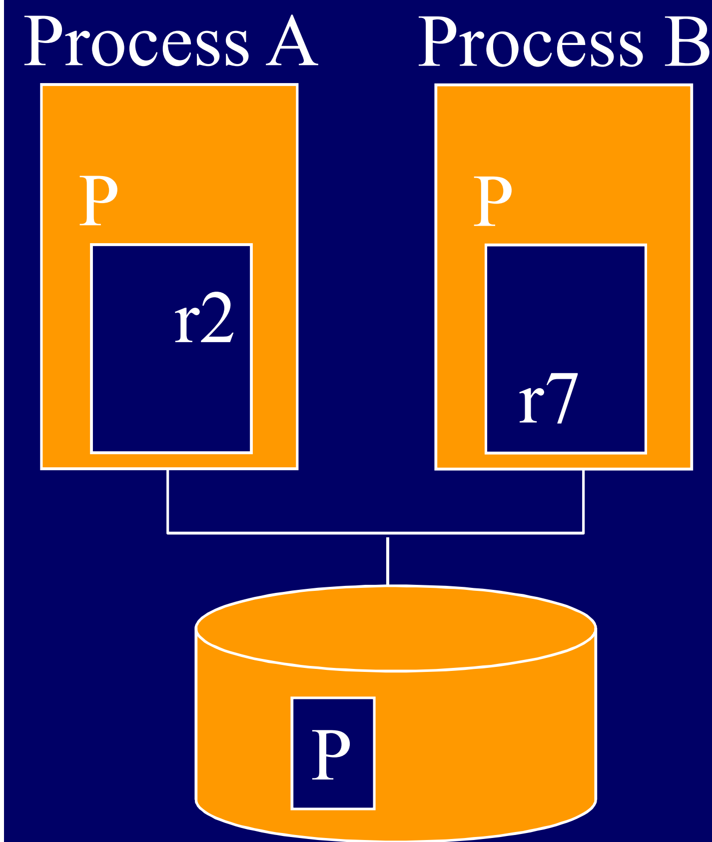
Logging B-Tree Operations

- To split a page
  - log records deleted from the first page (for undo)
  - log records inserted to the second page (for redo)
  - they're the same records, so long them once!
- This doubles the amount of log used for inserts
  - log the inserted data when the record is first inserted
  - if a page has N records, log N/2 records, every time a page is split, which occurs once for every N/2 insertions

User-level Optimizations

- If checkpoint frequency is controllable, then run some experiments
- Partition DB across more disks to reduce restart time (if Restart is multithreaded)
- Increase resources (e.g. cache) available to restart program.

Shared Disk System



- Can cache a page in two processes that write-lock different records
- Only one process at a time can have write privilege
- Use a global lock manager
- When setting a write lock on P, may need to refresh the cached copy from disk (if another process recently updated it)

- Use version number on the page and in the lock

Shared Disk System

- When a process sets the lock, it tells the lock manager version number of its cached page.
- A process increments the version number the first time it updates a cached page.
- When a process is done with an updated page, it flushes the page to disk and then increments version number in the lock.
- Need a shared log manager, possibly with local caching in each machine.

4. Media Failures

- A media failure is the loss of some of stable storage.
- Most disks have MTBF over 10 years
- Still, if you have 10 disks ...
- So shadowed disks are important
  - Writes go to both copies. Handshake between Writes to avoid common failure modes (e.g. power failure)
  - Service each read from one copy
- To bring up a new shadow
  - Copy tracks from good disk to new disk, one at a time
  - A Write goes to both disks if the track has been copied
  - A read goes to the good disk, until the track is copied

RAID

- RAID - redundant array of inexpensive disks
  - Use an array of N disks in parallel
  - A stripe is an array of the i^{th} block from each disk
  - A stripe is partitioned as follows:

- Each stripe is one logical block, which can survive a single-disk failure.

Where to Use Disk Redundancy?

- Preferably for both the DB and log
- But <u>at least</u> for the log
  - In an undo algorithm, it's the only place that has certain before images
  - In a redo algorithm, it's the only place that has certain after images
- If you don't shadow the log, it's a single point of failure

Archiving

- An archive is a database snapshot used for media recovery.
  - Load the archive and redo the log
- To take an archive snapshot
  - write a start-archive record to the log
  - copy the DB to an archive medium
  - write an end-archive record to the log
    (or simply mark the archive as complete)
- So, the end-archive record says that all updates before the start-archive record are in the archive
- Can use the standard LSN-based Restart algorithm to recover an archive copy relative to the log.

Archiving (cont'd)

- To archive the log, use 2 pairs of shadowed disks. Dump one pair to archive (e.g. tape) while using the other pair for on-line logging. (I.e. ping-pong to avoid disk contention)
  - Optimization - only archive committed pages and purge undo information from the log before archiving
- To do incremental archive, use an <u>archive</u> bit in each page.
  - Each page update sets the bit.
  - To archive, copies pages with the bit set, then clear it.
- To reduce media recovery time
  - rebuild archive from incremental copies
  - partition log to enable fast recovery of a few corrupted pages