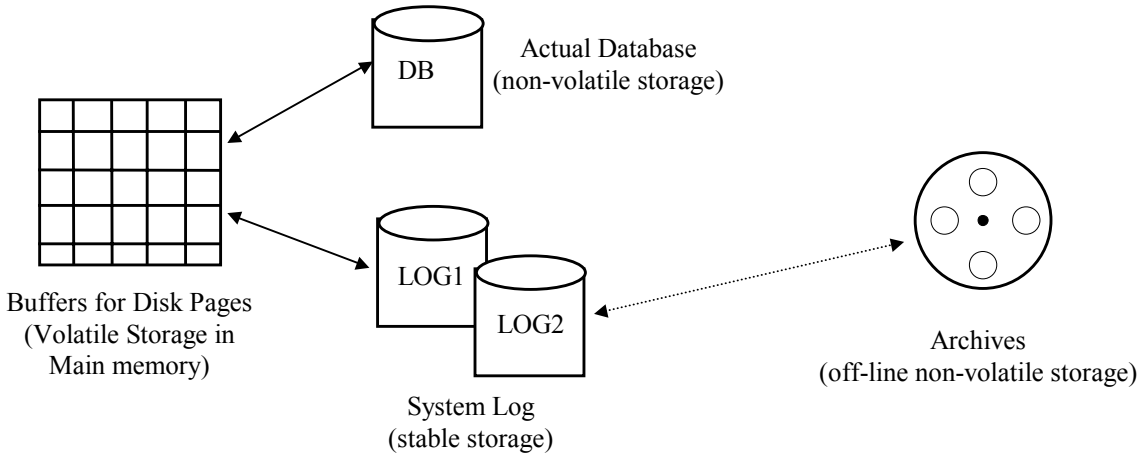
Recovery with Aries

DBMS Storage Types:



Assumptions about failures/crashes:

- **Volatile storage** (i.e. buffers in main memory) is lost when a crash occurs
- **Non-volatile storage** survives software crashes and only fails due to media failures (more reliable than volatile storage)
- **Stable storage** "never" fails. Talk about mirrored disks and RAID devices
- **Non-volatile off-line storage** also exists and is highly reliable (e.g. tape archives)

Possible Failure Types

- **Action failure** (bad action parameters, etc.).
- **Transaction failure** (deadlock, abort, local errors)
- **System failure** (serious error, hardware crash)
- **Media failure** (disk crash)

Goal: Always be able to:

- Back out effects of uncommitted transactions
- Recover results of committed transactions
- Get consistent snapshot of the DB (as a result of above)

Approach to achieving the goal:

- Some concurrency control mechanism such as locking with fancier tricks on "hot spots" such as indices.
- DO-UNDO-REDO paradigm for log records

- Write Ahead Log (WAL) protocol
- Two-Phase Commit protocol for distributed transactions spanning

To permit per-transaction UNDO/REDO every recoverable action (such as update record) must coded as four components.

- DO: perform the action, and record UNDO/REDO information in the log
- UNDO: undo the action using the UNDO information in the log
- REDO: redo the action using the REDO information in the log
- Display: translate actions's log entry into human readable form

Typical DBMS (ARIES or otherwise) log record entry includes:

- LSN (generally implicit)
- Record ID/Name (globally unique)
- Old record value (for UNDO)
- New record value (for REDO)

Provided by Action

- Transaction ID
- Action ID (e.g. update, delete, insert, …)
- Log record length
- LSN of previous log record generated by this XACT

Provided by Log Manager

Question: What happens if UNDO/REDO operations are in progress during a crash?

Answer: UNDO and REDO must be idempotent. Thus, f(DB) = f(f(DB)) = f(f(f(DB))), …. Also, must be able to "UNDO" operations that never happened (in non-volatile storage) and to REDO operations that did take place already (LSN's used to do this)

Write Ahead Log Protocol (WAL):

Question: When must UNDO information reach the log?
Question: When must REDO information reach the log?
Answer: WAL protocol means that:

- UNDO information for an update must reach the log before the update is applied to the non-volatile copy of the DB. Explain why!

- REDO information must reach the log before the commit record for the transaction gets there (i.e. before we promise not to lose its updates). Likewise, for UNDO information, so we can go either way for the two-phase commit (distributed DBMS only).

Log Sequence Number (LSN) useful:

- Every log record has an associated LSN. The LSN is really the address of the log record in the log.

- Every recoverable object (normally a page) has a "high water mark" which is the largest LSN that applies to it. In ARIES this is called the PageLSN.

- Use LSN/High Water Marks (HWM) as follows:

  ⇒ On update, set HWM of object to the LSN of the log record corresponding to the update operation.

  ⇒ Don't allow an object to be written out to non-volatile storage before the log has been written past the object's HWM

  ⇒ Get idempotent UNDO/REDO operations by simply checking HWM versus LSN to see whether or NOT UNDO/REDO is necessary.

Sumary Thus Far:

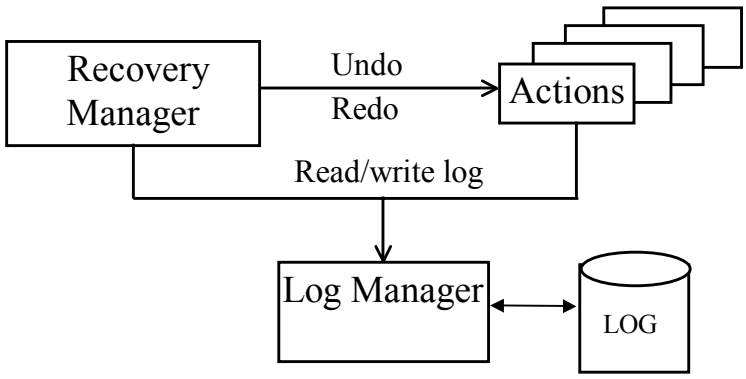
- Do/Undo/Redo - Allows for recovery of actions
- WAL - makes sure that Undo/Redo is always really possible

Recovery Manager Structure

Recovery management has two components:

- Recovery manager - Keeps track of transactions, handles commit and abort for transactions, takes care of system checkpoint and restart

- Log manager - Provides log service to the recovery manager and other components that may need its services



- **Reminder:** Recovery manager really has two jobs:

    1. Handle per transaction recovery (UNDO) when "minor" errors occur
    2. Get DB back to most recently committed state when a "serious" error occurs (i.e. a crash)

Other Recovery-Related Concepts/Issues:

Flavors of Recovery Schemes [Haerder/Reuter Surveys Paper]
Can combine features of:

Atomic/ ~Atomic	Can updated pages be grouped in all/nothing way for disk I/O (really shadows vs. logging at least in practice)
Force/~Force	Changes forced to disk at commit time?
Steal/~Steal	OK for uncommitted data to migrate to disk?

Logging schemes can be:

Physical	e.g., did this to these bytes on this specific page (log before + after + physical location information
Logical	e.g. changed field 2 of record with this RID. Log old/new field value plus information on logical entry affected. Might imply the index changes needed, and records may move, etc.

ARIES

- WAL based
- Supports ~atomic/~force/steal
- Support advanced locking techniques:
    - ⇒ Records
    - ⇒ Semantic operations (e.g. increment/decrement field values)
    - ⇒ Non-2PL index locking methods
- Also supports
    - ⇒ Flexible storage (i.e. space management)
    - ⇒ Partial rollbacks (i.e. save units)
    - ⇒ Recovery independence (media/page-change errors)
    - ⇒ Parallelism & low overhead
- And, importantly:
    - ⇒ Bounded space consumed with repeated crashes

ARIES is the state-of-the-art. So you can learn lots by understanding it. Also, the paper has a great discussion of stuff like:

- Latches
- Conditional vs. unconditional locks
- Lock durations (instant, manual, and commit)

ARIES basics

Each page in the database has a pageLSN field which identifies which log record is associated with its latest update. Uses of the pageLSN include:

- During recovery: Does a given update need to be redone? Not if pageLSN is bigger than the log record for the update

- During analysis: Where does the REDO pass have to start? Based on information on LSN's of resident and/or possibly resident dirty pages

Each XACT has a reverse-chained list of log records formed using the prevLSN field of the log record to point to the previous log record generated by the transaction.

Important note: the pageLSN for a page is always a monotonically increasing value!!

- On Undo, ARIES logs a "compensation" log record (or CLR), and its LSN goes on the page as you will see. Otherwise, undoing an update would require that the pageLSN be rolled back to its previous value

Bottom line:

- Physical REDO ("repeat history")
- Logical UNDO

Normal Operation

Transactions do logging of stuff using WAL rules as discussed earlier.

Two key data structures for ARIES:

1. **XACT Table**: This table has entries for every active transaction, keeping the LSN of the last log record generated by the transaction (known as the lastLSN field).

2. **Dirty Page Table** (DPT): has entries for all buffer-resident dirty pages, where dirty means changed but not on disk yet. Entries contain an recoveryLSN field which is the LSN of the log record that first (since last write) dirtied the page. Therefore,

recoveryLSN for page P is the address of the earliest log record relevant to recoverying page P!

Operations keep these structures updated in the appropriate way. In addition, checkpoints are performed on a periodic basis to help avoid excessively long recovery times. Do this as follows:

- Fuzzy checkponts - don't stop transaction processing and don't force any pages out to the database.

- Checkpoint record contains:

  1. XACT table (i.e. what transactions are running at the time of the checkpoint)
  2. Dirty Page Table (DPT)

- Actually the checkpoint is performed as a beginChkpt/endChkpt pair (to avoid quiescing operation); Stuff in between taken into account when initializing world (at recovery's start) from checkpoint.

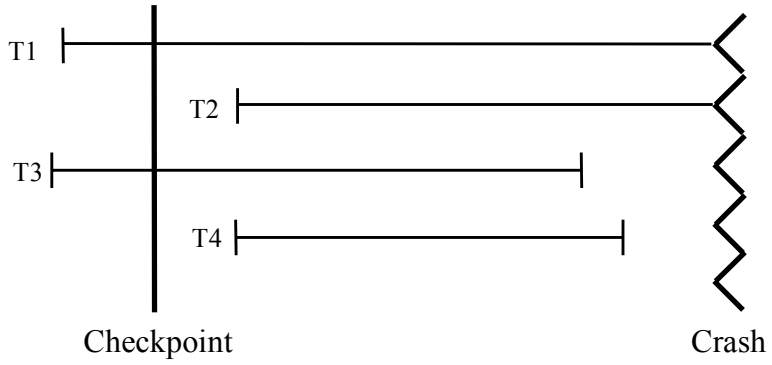
- The LSN of the Checkpoint record is then written at a well-known location on stable storage. This is written (atomically) after checkpoint finishes successfully.

Question: Is this enough to ensure fast recovery?

Answer: no, you also need to periodically write out old (especially hot) dirty pages! See why? We will assume some background process does this. The general rule of thumb is that any dirty page in memory that has not been written to disk for two consecutive checkpoints gets flushed at the second checkpoint. However, ARIES does not strictly require this.

Crash Recovery

Crash recovery is performed after a crash occurs. Basically, recovery must insure that the effects of committed transactions are reflected on non-volatile storage and effects of uncommitted transactions on non-volatile storage at the time of the crash must not persist after restart. Consider the following picture:



In this picture T1 and T2 are "losers". Their effects are undone during the recovery process. T3 and T4 are "winners". Recovery must insure that the effects of T3 and T4 are reflected on non-volatile storage when the system is restarted. The purpose of the analysis phase is to figure out which transactions are winners and losers. A key point to observe is that T2 and T4 were not active as of the most recent checkpoint record.

Three passes over the log are performed:

1. **Analysis pass**: This pass figures out information about dirty pages and uncommitted transactions. It starts at the most recent checkpoint, going forward

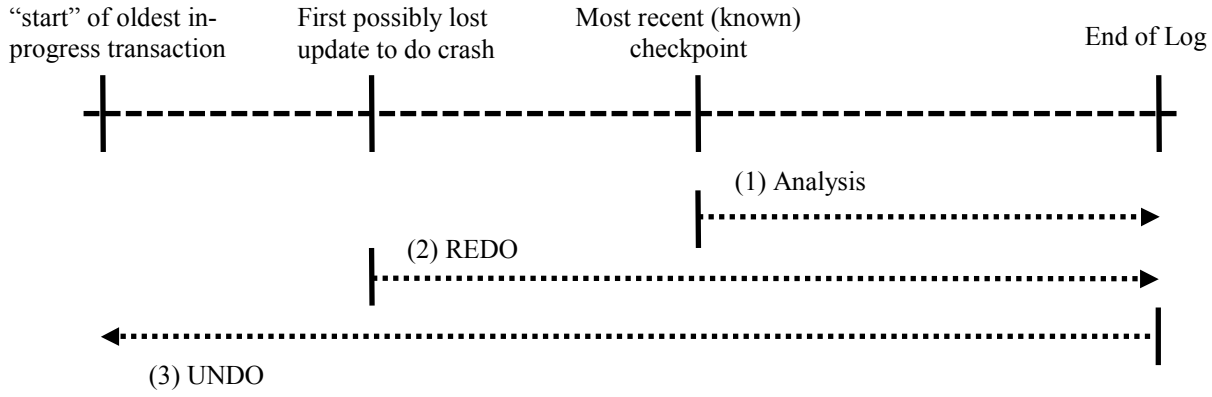
2. **REDO pass**: This pass goes forward, redoing updates from the earliest spot in the log where an update might have been lost. Idea is to "redo history" to ensure that all logged operations have been applied prior to the Undo pass. A key idea of ARIES is that all logged updates are redone, not just the ones from the committed transactions.

3. **UNDO pass**: This pass goes backward from the end of the log, removing the effects of all uncommitted updates from the DB.

Question: Why REDO history?

Answer: It turns out to (greatly) simplify otherwise complex problems raised by fine-grained locking, space management, index locking, etc.

Thus, we have, for ARIES:



Note: "Start", above, means the oldest active transactions first update log record

Analysis Pass

Its jobs are to:

- Determine starting point for REDO pass
- Identify set of "might have been dirty at crash pages", to avoid unnecessary I/O during REDO.
- Identify "Loser" transactions ie. uncommitted, active transactions at the time of the crash which must be dealt with in UNDO pass.

How it works:

- Scan log forward from most recent checkpoint record
- Initialize XACT Table and Dirty Page Table (DPT) to their respective states in the checkpoint record
- Process subsequent log records, updating XACT Table and DPT appropriately based on the log records' contents:

    ⇒ Add/Remove XACTS as the come (begin transaction) and go (commit)
    ⇒ Add entries to the DPT for additional updated pages

When analysis pass is done:

- DPT = conservative estimate of possible "dirty-at-crash" pages.

  Why conservative? Because the page might have actually gone to disk after the checkpoint but no way to know for sure. In earlier recovery schemes (e.g. Lindsay notes) the log contains read/write page log records, enabling one to remove entries from DPT during the analysis phase.

- XACT Table = Exact list of transactions requiring UNDO (due to being uncommitted at the time of the crash)

- Earliest recoveryLSN in DPT = firstLSN = place to being the REDO pass

REDO Pass

The job of the REDO pass is to put the database back in the same (physical) state it was in immediately before the crash occurred. The REDO phase, in effect, "repeats" history.

How it works

- Scan log forward from firstLSN (determined by analysis)

- For each log record, REDO the operation by:
    1. reapply the logged update to the page it applies to
    2. set the pageLSN of the page to that of the log record

- When do you actually have to perform the REDO work specified by the log record

    $\Rightarrow$ Don't have to REDO if $LSN_{LogRec} \leq$ pageLSN of the affected page (since the page indicates it was already done!)

    $\Rightarrow$ Don't even have to check pageLSN if the page $\notin$ DPT or if the page's recoveryLSN in DPT > LSN_{LogRec} (since DPT $\Rightarrow$ it was already done in both these cases)

UNDO Pass

The job of the UNDO pass is to provide atomicity by removing "all" stuff done by uncommitted transactions. Due to the repeating history of ARIES, UNDO in ARIES is unconditional.

In some ways ARIES is kind of weird. During the REDO pass, ARIES redoes work on the behalf of "loser" transactions only to UNDO this work during the UNDO pass. This seems wasteful. Earlier recovery schemes (as developed by the System R team), did the UNDO pass first and then the REDO pass. While this saved some work, it actually is more complicated with respect to index updates.

How UNDO works:

- Scan backward from the end of the log, undoing updates of uncommitted transactions.
- To undo an update:

    – Apply UNDO function associated with the action indicated in the log record

    – Write a CLR (compensation log record) to the log to indicate that UNDO occurred, with:

        * UNDO Information (what was done)
        * undoNextLSN = LSN of next (older) log record that must be undone for this transaction = prevLSN field of the log record being undone.

- When CLR's are encountered during UNDO:
  - Don't do anything to the page, ...
  - Just follow undoNextLSN pointer to the next previous log record needing to be applied for this transaction

Question: What good are CLR's then?

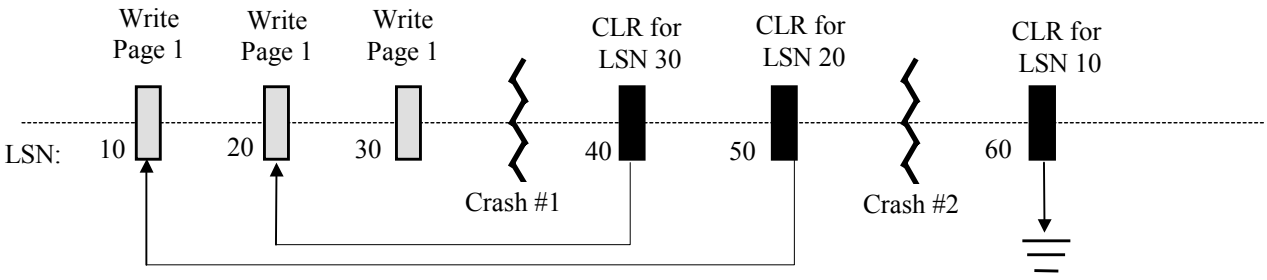
Answer: They serve several purposes:

1. Keep pageLSN's growing, even during recovery

2. Never have to UNDO an UNDO (which bounds the amount of logging at recovery time with repeated crashes during recovery). This is a big deal as you are doomed if you run out of log spaced during recovery.

- UNDO for an aborting transaction is similar to this

Example (log records for one transaction for one page, for simplicity):



Notes on the example:

- **1st crash outcome is:**
  - Analysis determines that this transaction is a loser
  - REDO: log records 10, 20, and 30 are redone (repeating history)
  - UNDO: undo 30, then undo 20, writing CLRs

- **2nd crash outcome is:**
  - Same restart analysis. The transaction must be undone
  - REDO: log records 10 to 50 are redone (as needed depending on the pageLSN of page 1)
  - UNDO: "UNDO" 50, then skip to UNDO 10, writing CLR

- No extra log records are generated during the second crash! (vs. having even more to do this time, as some other logging approaches might have.

Other Goodies

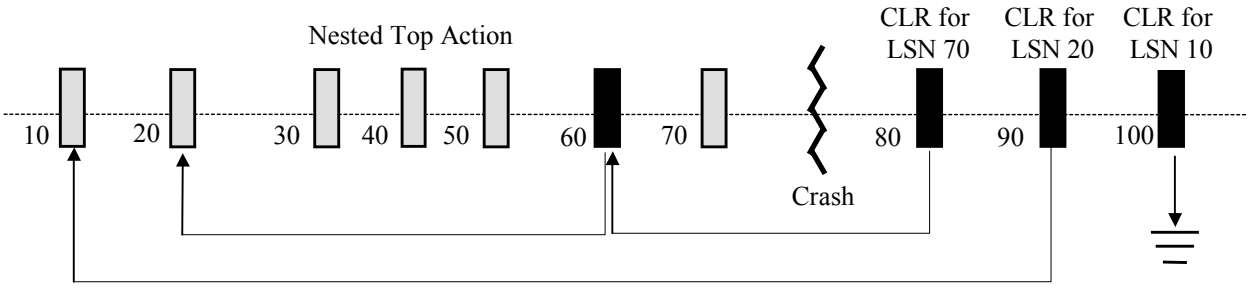
A "Biggie" is a feature called **nested top actions**, which ARIES supports nicely via CLRs! A nested top action is work that is done by a transaction that is unconditional. That is, the work is committed even if the transaction fails.

Examples:

- B+ tree page split
- Addition of space to a file

Other subsequent transactions depend on the durability of such changes if using high concurrency cc for such operations.

- Trick is to use a "Dummy" CLR to bracket these, hence preventing undo, i.e.



Do you see how/why this works???

Summary of ARIES

Basic approach is:

- Analysis + REDO (repeat history) + UNDO (loser transactions)
- REDO is physical, undo is logical, CLR's used

Results algorithm is extremely nice/flexible:

- Permits ~atomic/~force/steal buffering
- One pageLSN per page (monotonically increasing)
- Fine-grained locking and fancy semantic stuff supported (key-range locking)
- Nested top actions (efficient!) for indices and space managment

- Bounded logging regardless of number of failures during restart
- Low-cost, fuzzy checkpointing
- Recovery independence (roll each page forward using log)