Transactional Recovery
Transactions: ACID Properties

“Full-blown” transactions guarantee four intertwined properties:

- **Atomicity**. Transactions can never “partly commit”; their updates are applied “all or nothing”.
  
  The system guarantees this using logging, shadowing, distributed commit.

- **Consistency**. Each transaction $T$ transitions the dataset from one semantically consistent state to another.
  
  The application guarantees this by correctly marking transaction boundaries.

- **Independence/Isolation**. All updates by $T1$ are either entirely visible to $T2$, or are not visible at all.
  
  Guaranteed through locking or timestamp-based concurrency control.

- **Durability**. Updates made by $T$ are “never” lost once $T$ commits.
  
  The system guarantees this by writing updates to stable storage.
The Problem of Distributed Recovery

In a distributed system, a recovered node’s state must also be consistent with the states of other nodes.

E.g., what if a recovered node has forgotten an important event that others have remembered?

A functioning node may need to respond to a peer’s recovery.

- rebuild the state of the recovering node, and/or
- discard local state, and/or
- abort/restart operations/interactions in progress
e.g., two-phase commit protocol

How to know if a peer has failed and recovered?
**Key idea**: supplement the home data image with a log of recent updates and/or events.

- append-only
- sequential access (faster)
- preserves order of log entries
- enables *atomic commit* with a single write

Recover by traversing, e.g., “replaying”, the log.

Logging is fundamental to database systems and other storage systems.
Commuting Distributed Transactions

Transactions may touch data stored at more than one site.

Each site commits (i.e., logs) its updates independently.

*Problem*: any site may fail while a commit is in progress, but after updates have been logged at another site.

An action could “partly commit”, violating atomicity.

Basic problem: individual sites cannot unilaterally choose to abort without notifying other sites.

“Log locally, commit globally.”
**Two-Phase Commit (2PC)**

*Solution*: all participating sites must agree on whether or not each action has committed.

- **Phase 1**: The sites *vote* on whether or not to commit.
  
  *precommit*: Each site prepares to commit by logging its updates before voting “yes” (and enters *prepared* phase).

- **Phase 2**: Commit iff all sites voted to commit.
  
  A central transaction *coordinator* gathers the votes. If any site votes “no”, the transaction is aborted.
  
  Else, coordinator writes the *commit* record to its log.
  
  Coordinator notifies participants of the outcome.

*Note*: one server ==> no 2PC is needed, even with multiple clients.
The 2PC Protocol

1. \(Tx\) requests commit, by notifying coordinator (\(C\))
   \(C\) must know the list of participating sites.

2. Coordinator \(C\) requests each participant (\(P\)) to prepare.

3. Participants validate, prepare, and vote.
   Each \(P\) validates the request, logs validates updates locally, and responds to \(C\) with its vote to commit or abort.
   If \(P\) votes to commit, \(Tx\) is said to be “prepared” at \(P\).

4. Coordinator commits.
   Iff \(P\) votes are unanimous to commit, \(C\) writes a commit record to its log, and reports “success” for commit request. Else abort.

5. Coordinator notifies participants.
   \(C\) asynchronously notifies each \(P\) of the outcome for \(Tx\).
   Each \(P\) logs the outcome locally and releases any resources held for \(Tx\).
Handling Failures in 2PC

How to ensure consensus if a site fails during the 2PC protocol?

1. A participant $P$ fails before preparing.
   
   Either $P$ recovers and votes to abort, or $C$ times out and aborts.

2. Each $P$ votes to commit, but $C$ fails before committing.
   
   Participants wait until $C$ recovers and notifies them of the decision to abort. The outcome is uncertain until $C$ recovers.

3. $P$ or $C$ fails during phase 2, after the outcome is determined.
   
   Carry out the decision by reinitiating the protocol on recovery.
   
   Again, if $C$ fails, the outcome is uncertain until $C$ recovers.
Achieving Atomic Durability

Atomic durability dictates that the system schedule its stable writes in a way that guarantees two key properties:

1. *Each transaction’s updates are tentative until commit.*
   
   Database state must not be corrupted with uncommitted updates.

   If uncommitted updates can be written to the database, it must be possible to *undo* them if the transaction fails to commit.

2. *Buffered updates are written to stable storage synchronously with commit.*

   **Option 1:** *force* dirty data out to the permanent *(home)* database image at commit time.

   **Option 2:** commit by recording updates in a *log* on stable storage, and defer writes of modified data to home *(no-force).*
**Atomic Durability with Force**

A *force* strategy writes all updates to the home database file on each commit.

- must be synchronous
- disks are block-oriented devices

**What if items modified by two different transactions live on the same block?**

- need page/block granularity locking
- writes may be scattered across file
  - poor performance

**What if the system fails in the middle of the stream of writes?**
Shadowing

Shadowing is the basic technique for doing an atomic force.
reminiscent of *copy-on-write*

1. starting point
   modify purple/grey blocks

2. write new blocks to disk
   prepare new block map

3. overwrite block map
   *(atomic commit)*
   and free old blocks

*Frequent problems:* nonsequential disk writes, damages clustered allocation on disk.
**No-Force Durability with Logging**

*Logging* appends updates to a sequential file in temporal order.

- **Durability**
  
  The log supplements but does not replace the home image; to recover, replay the log into the saved home image.

  The home image may be optimized for reads since there is no need to force updates to home on transaction commit.

- **Atomicity**
  
  *Key idea:* terminate each group of updates with a *commit record* (including transaction ID) written to the log tail *atomically.*

- **Performance**
  
  The log localizes updates that must be done synchronously, and so is well-suited to rotational devices with high seek times.

  *Drawback:* some updates are written to disk twice (log and home).
Anatomy of a Log

**physical**
Entries contain item *values*; restore by reapplying them.

**logical (or method logging)**
Entries contain *operations* and their arguments; restore by reexecuting.

**redo**
Entries can be replayed to restore committed updates (e.g., *new value*).

**undo**
Entries can be replayed to roll back uncommitted updates.
Redo Logging: The Easy Way

Simple Case: logging for a short-lived process running in a virtual memory of unbounded size.

1. Read the entire database into memory.
2. Run code to read/update in-memory image.
3. Write updates to the log tail and force the log to disk on each commit.
   
   write-ahead logging

4. Before the process exits, write the entire database back to home (atomically).

   e.g., CMU Recoverable Virtual Memory (RVM) or Java logging and pickling (Ivory)

   no-force no-steal

long-term storage (home)
Why It’s Not That Easy

1. We may need some way to undo/abort.
   
   Must save “before images” (undo records) somewhere. Maybe in the log? Or in a separate log in volatile memory?

2. All of those sluggish log forces will murder performance.

3. We must prevent the log from growing without bound for long-lived transactions.

   Checkpoints: periodically write modified state back to home, and truncate the log.

4. We must prevent uncommitted updates from being written back to home....or be able to undo them during recovery.

   How to do safe checkpointing for concurrent transactions?

   What about evictions from the memory page/block cache (steal)?
Fast Durability 1: Rio Vista

Idea: what if memory is nonvolatile?

- uninterruptible power supply (UPS)
  - $100 - $200 for a “fig-leaf” UPS
- durability is “free”
  - update-in-place; no need to log updates to disk
- atomicity is fast and easy
  - uncommitted updates are durable....
    ...so keep an undo log in memory, and discard it on commit
  - library only: no kernel intervention
- not so great for American Express

David Lowell/Peter Chen (UMich)
[ASPLOS96, SOSP97, VLDB97]
Fast Durability II: Group Commit

Idea: amortize the cost of forcing the log by committing groups of transactions together.

Delay the log force until there’s enough committed data to make it worthwhile (several transactions worth).

Accumulate pending commits in a queue: push to the log when the queue size exceeds some threshold.

• assumes independent concurrent transactions
  cannot report commit or release locks until the updates are stable

• transactions can commit at a higher rate
  keep the CPU busy during log force; transfer more data with each disk write

• transaction latency goes up
A Quick Look at Transaction Performance

*Figure of merit:* transaction throughput.

How many transactions per second (TPS) can the system commit?

Concurrency control and transaction overhead are factors, but performance is generally driven by I/O effects.

Fault-reads and writebacks if the database does not fit in memory.

Commit costs for durability.

How fast is your system?

**RVM:** determined by transaction length and log-force latency.

**RVM with group commit:** for small concurrent transactions, throughput is determined by log bandwidth: add more spindles.

**Rio Vista:** how fast can you copy the data to the undo log?
The Need for Checkpointing

First complication: How to prevent the log from growing without bound if the process is long-lived?

Periodically checkpoint: flush all modified objects back to long-term home.
  - truncate log after checkpoint

Recover by replaying the log into the last checkpointed state.

Issues:
1. Checkpoints must be atomic.
2. Checkpoints must not write uncommitted updates back to home.
Atomic Checkpointing: Example

1. starting point
   last checkpoint file is \textit{cpt0}
   ready to write file \textit{cpt1}

2. write new checkpoint
   create file \textit{cpt1}
   leave \textit{cpt0} undisturbed

3. truncate old checkpoint
   truncate \textit{cpt0}
   (an atomic operation in most operating systems)
How to Deal with *Steal*?

A commit protocol must consider interactions between logging/recovery and *buffer management*.

- Volatile memory is managed as a cache over the database. Typically managed in units of pages (*buffers*), sized to match the logical disk block size.
- Cache management policies may evict a *dirty* page or buffer.
- This may cause an uncommitted writeback to home.
- This kind of buffering policy is called *steal*.
- *One solution*: “pin/update/log” [Camelot]
Goals of ARIES

ARIES is an “industrial strength” buffer management and logging/recovery scheme.

- no constraints on buffer fetch and eviction
  - steal
    - support for long-running transactions
- fast commit
  - no-force
- “physiological” logging of complete undo information
- on-line incremental “fuzzy” checkpointing
  - fully concurrent with automatic log truncation
- fast recovery, restartable if the system fails while recovering
Introduction to ARIES

1. Every log record is tagged with a monotonically increasing Log Sequence Number (LSN).
   At recovery, log records can be retrieved efficiently by LSN.

2. Keep a transaction table in memory, with a record for each active transaction.
   Keep each transaction’s lastLSN of its most recent log record.

3. Maintain a backward-linked list (in the log) of log records for each transaction.
   (Write the transaction’s current lastLSN into each new log record.)

4. Each record in the log pertains to exactly one page, whose ID is logged as part of the record.
ARIES Structures

Log start/commit/abort events.

Redo/undo records pertain to pages, with page ID and entire contents.

Log contains a back-linked list of all records for a given transaction.

transaction lastLSN status
17 13 active
18 15 committing
.... .... ....

per-page state for dirty pages
recoveryLSN = earliest log record updating this page
pageLSN = latest log record updating this page

Dirty page list

Log contains a back-linked list of all records for a given transaction.

Page q descriptor

Memory buffer manager

Per-page state for dirty pages
recoveryLSN = earliest log record updating this page
pageLSN = latest log record updating this page
The Dirty Page List

ARIES maintains a table of descriptors for dirty pages.

- When a page is updated, save the LSN of the log record containing the update in the page descriptor’s `pageLSN`.
- If an update dirties a clean page, save the LSN in the page descriptor’s `recoveryLSN`.
  
  `recoveryLSN` names the oldest log record that might be needed to reconstruct the page during recovery.

- When a dirty page is cleaned (pushed or evicted):
  
  Mark clean and remove from dirty page list.
  
  Save its current `pageLSN` on disk, to help determine which updates must be reapplied on recovery.
ARIES Recovery: The Big Picture

1. Dirty pages are written out (mostly) at the buffer manager’s convenience (with prewrites for on-line checkpointing).
   
   The pageLSN saved on disk with each page is a timestamp giving the most recent update reflected in the home disk image.

2. Periodic fuzzy checkpoints write the dirty page list and transaction table (but nothing else) to stable storage.
   
   on-line, nonintrusive, efficient, etc.

3. On fuzzy checkpoint, truncate old log records.
   
   It is safe to discard all records older than the recoveryLSN of the oldest page in the dirty page list (this is firstLSN).

4. On recovery, use saved recoveryLSN and pageLSNs to minimize recovery time.
1. **Analysis.** Roll log forward and rebuild the transaction table and dirty page list, including $firstLSN$.

   - Scan log forward from the last fuzzy checkpoint.
   - The rebuilt dirty page list is a conservative approximation.

2. **Redo.** Starting at $firstLSN$, scan forward in the log and process all redo records.

   - “repeating history”
   - Skip/prune redo records that we can determine are not needed.

3. **Undo.** Roll back all updates made by uncommitted transactions, including those we just redid.

   - Follow backward chain of log records for each transaction that has no commit record in the log.
Redo Pruning

During the redo phase, determine whether each redo record is needed by examining its LSN:

Call the LSN of the current log record \textit{currentLSN}.

- Skip the record if \textit{currentLSN} contains a page that is not in the restored dirty list.
- Skip the record if the restored \textit{recoveryLSN} for the modified page is later than the \textit{currentLSN}.
- Skip the record if the modified page’s saved \textit{pageLSN} is later than \textit{currentLSN}. 
Redo Pruning: Explanation

Case 1: *currentLSN* updated a *P* not in the restored dirty list.

The latest checkpoint revealed that *P* had been written back to its home location and not updated again before the failure.

Case 2: the restored *recoveryLSN(P) > currentLSN*.

The latest checkpoint revealed that *P* may have been dirty at failure time, but the last unsaved update to *P* was after the current log record.

Case 3: *pageLSN(P) > currentLSN*.

*P* may or may not have been dirty at failure time, but the on-disk record for *P* says that the *currentLSN* update had been saved.
Evaluating ARIES

The ARIES logging/recovery algorithm has several advantages over other approaches:

- *steal/no-force* with few constraints on buffer management
  
  Steals act as incremental, nonintrusive checkpoints.

- synchronous “fuzzy” checkpoints are fast and nonintrusive

- minimizes recovery work
  
  makes forward progress in failures during recovery

- *repeating history* redo supports logical undo logging and alternative locking strategies (e.g., fine-grained locking)

But: ARIES requires WAL with undos, LSNs written with every page, and redo records restricted to a single page.

...and will it work in a distributed system?
Client/Server Exodus (ESM-CS)

ESM/CS is a client/server object database system, like Thor:

- Clients are serial processes with private buffer pools. All data updates (except recovery) are made in client caches, but clients contact server on transaction create.
- Server coordinates page-level locking with strict 2PL (roughly).
- Clients use byte-range (object) logging: log records are sent to the server one page at a time as they are generated.
- Clients use WAL with steal/force buffering.
- Server uses modified ARIES algorithm for checkpoint/recovery.

Note implicit goal: client log records are never examined or modified by the server during normal operation.
Client/Server ARIES

Clients maintain private buffer pools.

Clients use WAL object logging with \textbf{force}.

Server’s buffer pool may not reflect all logged updates.

- missing updates
- missing dirty bits

No central point for assigning LSNs, so they may not increase monotonically.

Server manages checkpoints.
Distributed ARIES

The basic ARIES algorithm must be modified to work in a client/server system such as ESM/CS.

1. The server receives an update record from a client before it receives the modified page and recognizes it as dirty.
   
   ! Server does not mark page dirty when an update is received.

   Server’s checkpointed dirty page list may be incomplete.

2. LSNs are assigned independently by clients: how to order records?
   
   Server does not reassign global LSNs for received log records.

   LSNs from “slow” clients may be skipped in the redo phase, or they may cause earlier updates with larger LSNs to be skipped.

3. Undo operations may need to be conditional since the server may not have all updates in its buffer pool.
Problem 1: the Dirty Page List

Problem: the naive ARIES analysis phase may fail to fully rebuild the “global” dirty page list.

The scenario:
- client logs update record $U$ for clean $P$
- server checkpoints dirty page list: $P$ is clean
- client sends page $P$ (e.g., with commit request)
- server marks $P$ dirty (in its in-memory dirty page list)
- client logs commit record
- crash: server skips $U$ on recovery
Reconstructing the Dirty Page List

**Solution**: exploit **force**-like buffering policy on clients.

**Force** is not strictly necessary here, but ESM/CS uses it to avoid “installation reads” for page writeback on the server.

Think of it as forcing the client to log a fuzzy checkpoint before committing a transaction.

- Log a **commit dirty list** of page IDs (and **recoveryLSNs**) of pages modified by the transaction.

  Do it at commit time, **before** the **commit** record.

- In the analysis phase, scan for client commit dirty lists appearing in the log after the server’s last fuzzy checkpoint.

  Supplement the dirty page list in the server’s checkpoint.
Conditional Undo

**Problem:** Pages dirtied by *uncommitted* transactions still might not be recognized as dirty.

This makes it impossible to “completely” redo history.

Undo operations in the ARIES undo phase may corrupt the database if it does not reflect the updates to be undone.

**Solution:** conditional undo.

Details “left as an exercise”....
Problem 2: The Trouble with PageLSN

Redo records may be skipped incorrectly during recovery if LSNs are not monotonically increasing.

In ESM/CS clients assign LSNs independently.

ARIES redo will skip an update \( U \) on a page \( P \) if (e.g.):

- \( LSN(U) < PageLSN(P) \) means \( P \) was pushed after update \( U \).

  A writes LSN 20 for \( P \)
  A commits and sends \( P \) with \( PageLSN = 20 \)
  server buffer manager pushes \( P \): \( PageLSN(P) = 20 \)
  \( B \) acquires \( P \) and writes LSN 10 \( (U) \) for \( P \)
Handling *PageLSN*

ESM/CS modifies the handling of PageLSN as follows:

- Clients and servers maintain an LRC (*Log Record Counter*) for each page $P$.
  
  $LRC(P)$ is always stored/cached/sent with $P$.

- Clients increment their local copy of $LRC(P)$ on each update to page $P$.
  
  $LRC(P)$ is monotonically increasing for each page $P$.

- Each redo record for $P$ includes $LRC(P)$.

- *PageLSN* becomes *PageLRC*

  Stamp each page with $PageLRC = LRC(P)$ when flushed to disk.

  Replace *PageLSN* check for redo with a *PageLRC* check.
**Problem 3: RecoveryLSN**

The LRC trick doesn’t solve the related problem with skewed `recoveryLSN`, e.g.:

- A writes LSN 20 for clean $P$
- A sends $P$ to server with $recoveryLSN(P) = 20$
- $B$ acquires $P$ and writes update $U$ for $P$ with LSN 10
- Server crashes

  - During analysis, server rebuilds $recoveryLSN(P)$.
    - maximum LSN for $P$ updates appearing after last checkpoint
  - Server redo skips $U$ because $LSN(U) < recoveryLSN(P)$. 
Handling RecoveryLSN

Solution: Use logical clocks to coordinate assigned LSNs to ensure a safe partial ordering.

- Client receives current *end-of-log LSN* piggybacked on every response from server.
  (including transaction initiate)
- Client resets local LSN counter to the new end-of-log LSN.
- Server updates end-of-log LSN on receiving log records from a client.

Assigned LSNs will not be consecutive or monotonic, *but*:

LSNs for updates to shared *P* are always later than the end-of-log LSN at the time the transaction was initiated, and therefore later than previous *recoveryLSN* for *P*. 
Evaluating ARIES for ESM/CS

CS-ARIES preserves the efficient recovery and flexible buffering of the centralized ARIES scheme.

flexible support for rollbacks and undos

Could it be simplified by server processing of log records as they arrive from clients?

e.g., as needed for a modified object buffer [Thor] for object-grained updates

(also could avoid force policy on clients)

But server pays a high cost to examine log entries.

Does it require page-grained locking?