Failure and Recovery

- Failure and inconsistency
  - transaction failures
  - system failures
  - media failures

- Principle of recovery
  - redundancy
  - DB can be protected by ensuring that its correct state can be reconstructed from some other information stored redundantly in the system

- Database recoverability
  - bring the stable DB to a consistent state, removing effects of uncommitted transactions and applying missing effects of committed ones
Recovery

- Types of storage media
  - volatile storage: fast, not survive system crashes
  - non-volatile storage
  - stable storage; information never lost (practically)

- Recovery
  - ideally, stable DB should contain, for each data object, the last value written by a committed transaction
  - practically, stable DB might contain values written by uncommitted transactions, or might not contain values written by committed transactions
  - why? 1) updating of uncommitted transactions
  - 2) buffering committed values in the cache

- Restart operation
  - transform the stable database state into the most recent committed state (by relying on data only in stable DB)
Logs and Transaction States

- **Log**
  - a representation of the execution history
  - physical log: consisting entries of the form \([T, x, v]\)
    --- enables the system to determine, for each \(x\),
    the last committed value of \(x\)
  - logical log: a higher level description
    --- fewer log entries, but higher complexity
    during recovery (context-sensitive)

- **Transaction states**
  - active, committed, aborted
  - transaction lists maintain the identifiers of the set
    of transactions that are active, committed, or aborted
  - these lists are stored as part of the log
Recovery Management

- Requests coming to data manager
  - read, write, commit, abort, restart

Design rules for Recovery Manager
  - undo rule: committed value must be saved before overwritten by uncommitted value in stable DB
  - redo rule: before commit, new values it wrote must be in the stable storage (DB or log)

Recovery activity
  - preparation: during normal operation
  - actual recovery: after failure

Preparation: log and checkpoint
Cache Manager

• Fetch and flush
  - use dirty bit for deciding flushing operation
  - flush: if the slot in cache is not dirty, do nothing otherwise, copy the value into stable storage
  - fetch: select a slot, using replacement algorithm if full (and flush if necessary), and copy the value into the slot, reset dirty bit, update cache directory

• When to flush?
  - depends on recovery strategy of the system
  - different recovery algorithms use different strategies
Restart Activity

- Classification of recovery algorithms
  - require both undo/redo
  - undo/no-redo, no-undo/redo, no-undo/no-redo
- Restart must be idempotent
  - any sequence of incomplete executions followed by a complete execution of restart has the same effect as just one complete execution
- Garbage collection of log entries
  - log requirement: if stable database does not contain the last committed value, it must be in the log
  - entry \([T, x, v]\) can be deleted if \(T\) is aborted or \(T\) is committed and other committed transaction wrote into \(x\) after \(T\)
Recovery Algorithms

- **Undo/redo algorithm**
  - most complicated of the four recovery algorithms
  - flexible in deciding when to flush
  - maximize efficiency during normal operation
    at the expense of less efficient recovery

- **Comparisons with other algorithms**
  - issues: log space, disk I/O, and recovery time
  - no-redo requires more frequent flush
  - uncommitted T is allowed to replace dirty slot
    for in-place update: undo may be necessary

- **Restart procedure**
  - process the log in two scans:
    1) backward scan for undoing uncommitted transactions
    2) forward scan for redoing committed transactions
Undo/Redo Recovery

A transaction T writes the value v to data object X. What will happen?

1) System fetches X if it is not already in cache.
   - record v in the log and in X’s slot C
   - no need for the cache manager to flush C immediately
   - recording an update into stable storage is determined by the cache manager

2) If cache manager replaces C, and either T aborts or system fails before T commits, undo is required

3) If T commits and system fails before C is flushed, redo is required
Restart Procedure

1. Discard all cache slots

2. Set redone and undone to $\emptyset$.
   They are to keep the track of data objects redone/undone.

3. Scan the log backward until $\text{redone} \cup \text{undone} = \text{entire DB}$
   or no more log entry remains to examine.

   For each log entry $[T, x, v]$, if $x \notin \text{redone} \cup \text{undone}$
   
   1) if $x$ is not in the cache, allocate a slot for it

   2) if $T$ is in the commit list, copy $v$ into $x$’s slot
      and set $\text{redone} := \text{redone} \cup \{x\}$

   3) otherwise ($T \in \text{abort} \cup \text{active}$ but $T \notin \text{commit}$)
      copy before-image of $x$ w.r.t. $T$ into $x$’s slot
      and set $\text{undone} := \text{undone} \cup \{x\}$

4. For each $T_i \in \text{commit}$, if $T_i \in \text{active}$, remove $T_i$
   from active list

5. Acknowledge the completion of restart
Checkpointing

To reduce the work of restart, certain information is saved in stable storage during normal operation of the system.

- restart is doing more work than necessary because it examines every record ever written in the log, even though most data objects contain the right value

- Task of checkpointing
  1) output log records and mark the log
  2) write after-images of committed updates or before-images of aborted updates in the stable DB

- Types
  1) commit-consistent checkpoint
     - two delays: i) active T to finish and ii) cache flush
  2) cache-consistent checkpoint: eliminate the first delay
     - leaving active T in a blocked state, flush dirty slots, and place marker at the end of the log
Undo/No-redo Recovery

The algorithm records all of T’s updates in the stable database before T commits.

- before adding T to the commit list, flush each data object updated by T, if it is in the cache

- Commit procedure

1. flush all updates made by T
2. add T to the commit list
3. acknowledge commitment of T to the scheduler
4. delete T from active list

- log records for update after-images are not necessary
- may increase response time and heavy I/O load
- if updates are not updated again by other transactions, the flush work is not wasted.
  It is wasted, however, if it is a hotspot data object
No-undo/Redo Recovery

To avoid undo, we must not record updates of uncommitted transactions in the stable database.

- when a data object is updated, it is not written in the cache; this happens only after T is committed
- consequently, when a new value is written to the stable database as a result of flush, it is guaranteed to be a committed value of the data object

- Write operation
  - it only appends \([T, x, v]\) to the log, not in the cache

- Read operation
  - it returns i) the value updated by itself, or ii) fetch it into cache and then read the value

- Commit operation
  - fetches all updates and writes after-images, and add T into the commit list; before-images are not necessary
No-undo/No-redo Recovery

To eliminate both undo and redo, all of T’s updates must be recorded in the stable database in a single atomic operation when T commits. How?

- **Shadowing**
  - two directories: 1) current directory, including committed updates only, and 2) working directory
  - swaps the two directories in an atomic action

- **Stable database**
  - it always contains the latest committed state
  - virtually no work is necessary to abort a transaction or to restart after failure
  - efficient restart with high cost during normal operation for indirect access, destroyed locality, etc.
    --- similar to a good insurance with high premium