Recovery Techniques and Assumptions

All the techniques assume the following:

- Failures are detectable.
- Write operations are atomic (i.e., execute either in its entirety or not at all).
- If this is not the case, we consider this failure as media failure.
- The scheduler sends operations to DM in an order which produces correct and strict executions.
- No media failure.
- The granularity of Writes that DM processes is the same as the granularity of the atomic Write supported by the hardware.

Undo/Redo Recovery Algorithm

The following types of log records are used:

Commit, Abort record: \([T_i, \text{commit}][T_i, \text{abort}]\)

Update record, \(U_i: [T_i, x, b, a, \text{old-LSN}(x), \text{prev-LSN}(T_i)]\)

- \(T_i\): the id of the transaction that issued the Write
- \(x\): the address of the block being modified and the offset and length
- \(b\): the before image of the modified portion of the block
- \(a\): the after image of the modified portion of the block
- \(\text{old-LSN}(x)\): the LSN of \(x\)'s buffer before this update
- \(\text{prev-LSN}(T_i)\): the LSN of the preceding log record of this transaction (null if it's the first)

Checkpoint record: \([C_P, \text{id}, Ac]\)

- \(Ac\): a list of the active transactions at checkpoint time.
Undo/Redo Operations

- **RM-Read**(\(T_i, x\))
  - return BM-Read\((x)\)
- **RM-Commit**(\(T_i\))
  - Append \([T_i, commit]\) to log and flush all log buffers
  - Send \(ack\) to the scheduler
- **RM-Write**(\(T_i, x, a\))
  - Fix\((x)\) in buffers
  - Append the record \([T_i, x, b, a, old-LSN(x) = LSN(x), prev-LSN(T_i)]\)
  - Set LSN\((x) = U_i\)
  - BM-Write\((x, a)\) and Unfix\((x)\)
  - \(ack(W_i)\) to the scheduler

Undo/redo Operations (cont'd)

- **RM-Abort**(\(T_i\))
  - Let \(U_i\) be the LSN of the most recent update record of \(T_i\), \(U_i : [T_i, x, b, a, old-LSN], prev-LSN]\)
  - While \(U_i = null\) do:
    - Fix\((x)\) in buffers
    - BM-Write\((x, b)\)
    - Set LSN\((x) = old-LSN\)
    - Unfix\((x)\)
    - \(U_i = prev-LSN\)
    - Append \([T_i, abort]\) to the log and send \(ack\) to the scheduler

Fuzzy Checkpointing: Stable-LSN

- We attach the Stable-LSN field to each buffer block.
- Stable-LSN is the LSN of the last record in the log buffer when the data item presently occupying the buffer was last fetched or flushed.
- Stable-LSN describes the time at which a block and its corresponding disk block in stable storage are the same.

<table>
<thead>
<tr>
<th>page id name</th>
<th>dirty Bit</th>
<th>fix count</th>
<th>block LSN</th>
<th>Stable LSN</th>
<th>buffer number</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>812</td>
<td>805</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>123</td>
<td>123</td>
<td>2</td>
</tr>
</tbody>
</table>

Fuzzy Checkpointing Algorithm

- Stop accepting new operations (active transactions are blocked).
- read CP-LSN (the LSN of the latest checkpoint).
- Scan the buffer pool and for each dirty buffer whose Stable-LSN is less than CP-LSN do:
  - Flush all log records whose LSN is less than the buffer's block-LSN.
  - Flush this buffer.
  - Set Stable-LSN of this buffer to be the CP-LSN of the new checkpoint record (that will be written at the end of this process).
Fuzzy Checkpointing Algorithm

- Starting from [CP-LSN, Ac], scan the log forward.
  - For each commit or abort record for Ti, do
    Ac = Ac - {Ti}.
  - For each update record for Ti, do Ac = Ac U {Ti}.
  - Append a checkpoint record to the log buffer with the active list Ac, and flush this buffer.
  - Write the new CP-LSN to the predefined location.
  - Resume normal execution.

Restart Algorithm with Fuzzy Checkpoint

- Set CL = ∅ and AL = ∅.
- Read the log backwards, until the penultimate checkpoint record is reached, and for each record do:
  - If [Ti, commit], then CL = CL U {Ti}.
  - If [Ti, abort], then AL = AL U {Ti}.
  - If [Ti, x, b, a, old-LSN, prev-LSN], then
    - If Ti ∈ CL then ignore this record else AL = AL U {Ti}
    - If Ti ∈ AL then
      BM-Write(x, b)
      LSN(x) = old-LSN(U)
      if prev-LSN = null then AL = AL - {Ti}
    - If it is the latest checkpoint record then ignore it.

How to avoid Unnecessary Writes?

- Store LSN’s in the block header. At restart, use the LSN to find out exactly which updates in the log have already been moved to disk.
  - No need to redo these updates!
Restart: Backward Scanning

- For each update record U: \([T, x, \text{operation}, \text{old-LSN}, \text{prev-LSN}]\) of an uncommitted transaction (aborted or active at system crash) do:
  - Read the block of x in main and examine the LSN(x).
  - if \(\text{LSN}(x) < \text{LSN}(U)\) /* the update described in U did not do nothing */
  - else if \(\text{LSN}(x) = \text{LSN}(U)\) /* U describes the operation on x */
    - \(\text{LSN}(x) = \text{old-LSN}(U)\);
    - undo(operation);
  - else /* \(\text{LSN}(x) > \text{LSN}(U)\) */
    - do nothing;
    - /* x contains an update by a log record U appearing after U; implies that the transaction that produced U must have committed */

Restart: Forward Scanning

- Start from the penultimate checkpoint record and proceed forward.
- For each update record U: \([T, x, \text{operation}, \text{old-LSN}, \text{prev-LSN}]\) of a committed transaction \(T_i\) examine the LSN(x).
  - if \(\text{LSN}(x) < \text{LSN}(U)\) /* the update described in U didn't redo(operation); make it to the stable storage. */
  - else if \(\text{LSN}(x) = \text{LSN}(U)\) /* the update described in U on x is already in the stable database. */
    - do nothing.
  - else /* \(\text{LSN}(x) > \text{LSN}(U)\) */
    - /* there should be another log record after U that describes the update on x. */

Discussion

- Strong interaction between concurrency and recovery systems.
- The locking granularity must be at least as coarse as the recovery granularity.

Example:
- If the recovery granularity is a block/page we can not have record level locking.
- If the recovery granularity is a record we can not have field level locking.

Undo/No-Redo Recovery Algorithm

- It never requires redoing an update.
- Basically the same as the previous algorithm except the commit operation.
- Commit(\(T_i\))
  - for each \(x\) updated by \(T_i\) flush \(x\)'s buffer to stable storage.
  - add a commit record to the log.
- Restart
  - Restart requires one (backward) scan through the log.
  - Update log records need not include the after images.
Checkpointing

- Updates of committed transactions are in stable storage.
- However, we still need checkpointing to ensure that the before image of a data item updated by an aborted transaction is in stable storage.
- Can checkpoints be eliminated by requiring RM-Abort($T_i$) to flush the before images of all data items updated by $T_i$?

Undo/No-Redo and Multiversion Concurrency

- All versions of a data item $x$ are linked together in the stable storage. New versions of $x$ created by active transactions are added at the head of the list.
- Each version created by some $T_i$ is tagged by the $ts(T_i)$.
- No need to store the before image of $x$ in the log. It can be found in $x'$s list.
- The log consists of three lists: commit, abort, and active.
- On Restart, any version of some $x$ created by an aborted or active transaction is removed from $x'$s list.

A Variation that Eliminates Restart

- Every time a Read on $x$ is performed, $x$'s tag is examined.
  - If the transaction that created this version of $x$ is in the commit list, then this is a committed version of $x$.
  - If it is not, discard this version (here is the undo) and repeat this step with the next version of $x$.
- Useful idea when frequent system failures are anticipated.

No-Undo/Redo Recovery Algorithm

Updates of active transactions are not applied to the data items; instead, they are recorded in the log.

- RM-Write($T_i, x, a$)
  - Write a $[T_i, x, a]$ to the log.
- RM-Read($x$)
  - If $T_i$ had updated $x$ then return the after image of $x$ from the log.
  - Otherwise, return BM-Read($x$).
- Commit($T_i$)
  - Force-Write $[T_i, commit]$ to log (now, $T_i$ commits).
  - For each $x$ updated by $T_i$, issue BM-Write($x, a$).
- Abort($T_i$)
  - Ignore it (Send ack to the scheduler).
No-Undo/No-Redo Recovery Algorithm (Shadowing)

- Data items are referred indirectly by symbolic names.
- The actual location of each data item is stored in a directory.

When a transaction $T_i$ updates a data item $x$, it creates a new version of $x$ in stable storage and it records this update in a directory local to $T_i$.

Discussion

- Very fast Restart operation.
- Indirect addressing is more expensive than direct addressing (except if directory is small so it can be kept in main memory).
- Garbage collection of uncommitted transactions becomes difficult.
- Any physical arrangements of data items on disk is destroyed.
- Recovery from media failures is not addressed.

Commit

- Updated pages cannot be written to disk before Commit
  - No steal
  - Assume pin/unpin protocol with Buffer Manager
  - If allowed to write to disk before commit ➔ Steal

- All updated pages are immediately written to disk when a transaction Commits
  - Force
  - Otherwise ➔ No force
Summary of Recovery Strategies

- No-force and steal: redo/undo
  Best from performance point of view, if done correctly

- Force and steal: no-redo/undo
  Increased commit processing overheads, low restart overheads

- No-force and no-steal: redo/no-undo
  Intention lists -- higher normal processing overheads

- Force and no-steal
  Shadows -- higher space overheads, difficult for semantics-based concurrency control

Overview of Recovery Concepts