CS 448 Database Systems

Recovery Managers

Recovery Managers

- Depending upon whether or not undo, redo operations may be needed, there are four types of RMs: *Undo/Redo; No-Undo/Redo; Undo/No-Redo;* and *No-Undo/No-Redo*.
- We will discuss each of these at a high-level.
- Each RM can be specified by the following operations: *RM-Read; RM-Write; RM-Commit; RM-Abort;* and *Restart*.

Undo/Redo

- Most complex; and most flexible about cache management ➔ more efficient.
- Designed to perform well for normal operation. Is slower than others when handling failures.
- Decisions about when to fetch/flush the cache are left solely to the cache manager.
- Assume that log is a sequence of $[T_i,x,v]$ entries.
- Assume that the initial value of each data item is in the log.
- Log updates and commit list are always stable.

RM-Write

RM-Write(T_i,x,v)

1. Add T_i to the *active list*, if it's not already there
2. If x is not in the cache, fetch it.
3. Append *[T_i,x,v]* to the log.
4. Write v into the cache slot occupied by x.
5. Acknowledge the processing of *RM-Write(T_i,x,v)* to the scheduler.

RM-Read

RM-Read(Ti,x)

1. If x is not in the cache, fetch it.
2. Return the value in x's cache slot to the scheduler.

RM-Commit

RM-Commit(Ti)

1. Add T_i to the *commit list*.
2. Acknowledge the commitment to the scheduler.
3. Delete T_i from the *active list*.

RM-Abort

RM-Abort(T_i)

1. For each data item x updated by T_i:
   1. If x is not in the cache, allocate a slot for it;
   2. Copy the before image of x wrt T_i into x's slot.
2. Add T_i to the *abort list*.
3. Acknowledge the abortion of T_i to the scheduler.
4. Delete T_i from the *active list*.

Restart

Restart

1. Discard all cache slots.
2. Let *redone* ← {}; *undone* ← {};
3. Start with the last entry in the log and scan backwards. Repeat until *{redone U undone}* equals the set of all data items, or the log is empty:
   - For each log entry $[T_i, x, v]$, if x is not in *{redone U undone},* then
     - If x is not in the cache, allocate a slot for it.

Restart

Restart(contd.)

- If x is not in the cache, allocate a slot for it.
- If T_i is in the commit list, copy v into x's cache slot and set *redone* ← U {x};
- Otherwise (I.e., T_i is in the *abort list* or in the *active list* but not in the *commit list*), copy the before image of x wrt T_i into x's cache slot and set *undone* ← U *{x}.*

4. For each T_i in the *commit list*, if T_i is in the *active list*, remove it from there.
5. Acknowledge the completion of Restart.

Issues

- Assume that value written is different from current value; RM pins and unpins to ensure atomicity of reads and writes.
- RM-Commit: adding to commit list defines commitment.
- RM-Abort: Before-image only restored in cache, will migrate to stable DB through CM replacement.
- Restart: as above – values only restored in cache.
- The before-image can be found from the log.
- No new txns until RM acks end of restart.

Undo/Redo Rules

- The *Undo rule* is satisfied because before overwriting the final committed value in stable DB, the log record is written. This record is not garbage collected since it represents the final committed value.
- Since the log is assumed to be immediately stable, the *Redo rule* is also satisfied.

Checkpointing

- The restart procedure may potentially need to read the entire contents of the log.
- Much of the work may be unnecessary because changes have already been saved or undone.
- We can reduce the amount of work that restart has to do by periodically ensuring some guarantees about the state of stable storage and the cache: Checkpointing.

Checkpointing

- Mark the *log, commit list*, and *abort list* to indicate which updates are already written or undone in stable DB
  - Tells restart what undo/redo operations are unnecessary.
- Write the after images of committed updates or before images of aborted updates in the stable DB.
  - Do the work of restart during checkpointing, therefore speeding up recovery (at the cost of normal operation).

Commit-Consistent Checkpointing

1. Stop accepting new transactions
2. Wait for all active txns to terminate
3. Flush all dirty cache slots
4. Record in the log that a checkpoint has been done.

- Drawback: Slow
  - don't allow new txns until checkpoint is finished
  - a currently long running txn could really hold everything up
  - flushing all dirty slots is slow.

Commit-Consistent Checkpointing

- Recovery begins at the end of the log, working backwards.
- Undo/redo is done *as before*.
- Upon reaching a checkpoint – no more undo/redo needs to be done.
- May have to scan the log beyond checkpoint to determine appropriate before images for objects overwritten after the ckpt by non-committed txns.

Cache Consistent Checkpointing

1. Periodically, stop accepting new operations at RM.
2. Flush all dirty cache slots
3. Place markers at the end of log, and abort list

Restart

- As before work back from the end;
- All updates by committed txns done before the last checkpoint are stable and need not be redone.
- *All updates* of txns that *aborted before* the last checkpoint need not be undone.
- Need to redo *updates* by committed txns *that follow* the checkpoint.
- Need to undo all updates by active (at crash) txns and also all updates by txns that aborted after the checkpoint.

Fuzzy Checkpointing

- Cache consistent checkpoint can be further improved by reducing the amount of data flushed.
- Flush only those slots that have not been flushed since the previous (penultimate) checkpoint.
- Expect that normal cache replacement will have more time to flush dirty cache slots ➔ quicker checkpoint.
- Of course, now recovery has to go back a little further!

Recovery

- Restart redoes just those updates of txns in commit list that come after the penultimate checkpoint
- Undoes the updates of those txns that are either in the active (but not commit) list, or are in the abort list and follow the penultimate checkpoint marker in the abort list.