CS 2550 / Spring 2006
Principles of Database Systems

14 – Recovery Algorithms

Alexandros Labrinidis
University of Pittsburgh

Recovery Techniques and Assumptions

- Undo/Redo Algorithm
- Undo/No-Redo
- No-Undo/Redo (also called *logging with deferred updates*)
- No-Undo/No-Redo (also called *shadowing*)

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 executions that are correct and strict
- No media failure
- The granularity of Writes that DM processes is the same as the that 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, commit]
[T_i, abort]

Update record, U_i : [T_i, x, b, a, old-LSN(x), 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
- old-LSN(x): the LSN of x's buffer before this update
- prev-LSN(T_i): the LSN of the preceding log record of this transaction (null if it's the first)

Checkpoint record: [CP_{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 U_i to the log buffer: [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 $U_i \neq$ 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.

page Id name	dirty Bit	fix count	block LSN	Stable LSN	buffer number
x	0	0	812	805	0
y	1	2	10	7	1
2	0	1	123	123	2
⋮	⋮	⋮	⋮	⋮	⋮

Buffer 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 T_i, do $Ac = Ac - \{T_i\}$.
  - For each update record for T_i do $Ac = Ac \cup \{T_i\}$.
- 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 $CL = \varnothing$ and $AL = \varnothing$.
- Read the log *backwards*, until the *penultimate* checkpoint record is reached, and for each record do:
  - If [T_i, *commit*], then $CL = CL \cup \{T_i\}$.
  - If [T_i, *abort*], then $AL = AL \cup \{T_i\}$.
  - If [T_i, *x, b, a*, old-LSN, prev-LSN], then
    - If $T_i \in CL$ then ignore this record else $AL = AL \cup \{T_i\}$
    - If $T_i \in AL$ then
      BM-Write(*x, b*)
      LSN(*x*) = old-LSN(*U*)
      if prev-LSN = null then $AL = AL - \{T_i\}$
  - If it is the latest checkpoint record then ignore it.

Restart Algorithm with Fuzzy Checkpoint

- Add in *AL* all T_i's in *Ac* of penultimate checkpoint but not in *CL*.
- Proceed backwards, until $AL = \varnothing$, and for each update record *U*
  - If T_i in *AL* then
    issue BM-Write(*x, b*).
    if prev-LSN(*U*) = null then $AL = AL - \{T_i\}$.
- Starting from the penultimate checkpoint record proceed *forward*
  - For each update record such that $T_i \in CL$ issue BM-Write(*x, a*).
- Send *ack* to the scheduler.

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*, operation, old-LSN, 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 LSN(x) < LSN(U)    /* the update described in U did not
    do nothing                make it to stable storage */
else if LSN(x) = LSN(U)    /* U describes the operation on x */
    LSN(x) = old-LSN(U);
    undo(operation);
else /* LSN(x) > 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,* operation, old-LSN, prev-LSN] of a committed transaction T_i, examine the LSN(*x*).

```
if LSN(x) <  LSN(U)            /* the update described in U didn't
    redo(operation);              make it to the stable storage. */
else if LSN(x) = LSN(U)  /* the update described in U on x is
    do nothing.                   already in the stable database. */
else /* LSN(x) > LSN(U) */     /* there should be another log record
    do nothing;                   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.

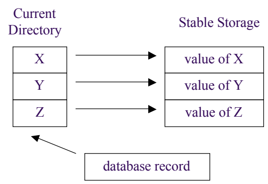
No-Undo/Redo Recovery Algorithm

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).

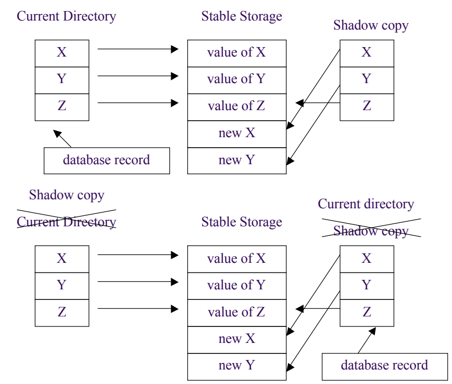
No-Undo/No-Redo Recovery Algorithm (Shadowing)

- 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.

Commit



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.

Force/Steal

- 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