Recovery Techniques

The System Failure Problem

- It is possible that the stable database: (because of buffering of blocks in main memory)
  - contains values written by uncommitted transactions.
  - does not contain values written by committed transactions.
- Recovery protocols implement two actions:
  - *Undo* action: required for atomicity.
    - Undoes all updates on the stable storage by an uncommitted transaction.
  - *Redo* action: required for durability
    - redoes the update (on the stable storage) of committed transactions.

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*)

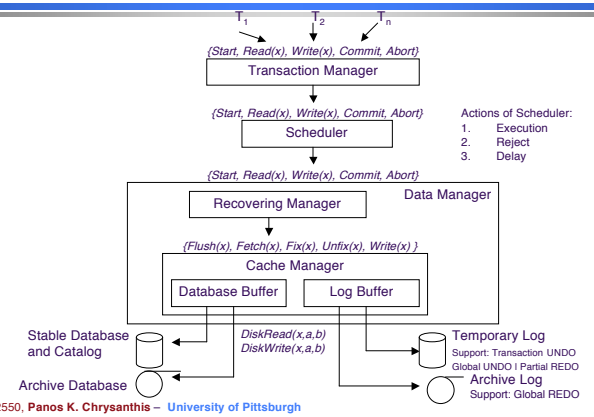
Failure Types

- **Program Failures**
  - logical errors
  - bad input
  - unavailable data
  - resource limits
  - user cancellation
- **System Failures**
  - computer hardware malfunction
  - bugs in O.S.
  - power failures
  - operator error

Failure Types

- **Media Failures**
  - disk head crash
  - data transfer error
  - Disk controller failure
- **Unrecoverable errors**
  - failure to make archive dumps
  - destruction of archives

Centralized DBMS



Recovery Techniques and Assumptions

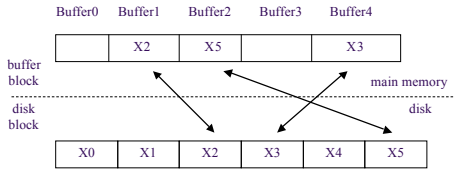
All the techniques assume the following:

- 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 one of the atomic Write supported by the hardware.

Buffer Management

The goal of Cache or Buffer Manager (*BM*) is to maximize the likelihood that a block of data needed by a transaction is in main memory.

- The main memory is partitioned into *buffer blocks*(or simply blocks).
- The size of a buffer is equal to the *disk block* size.



Buffer Management

- If no more buffers are available the BM must *replace* one of the buffer blocks (writing the block back to disk, if it has been updated).
  - Least Recently Used (LRU),
  - Least Frequently Used (LFU), etc.
- Concurrency and recovery are two other factors affecting the replacement algorithm.

Buffer Management Table

page Id	dirty bit	fix count	buffer number
x	0	0	0
y	1	2	1
z	0	1	2

- Buffer Management Operations
  - Read, Write, Fetch, Flush, Force-Write, Fix, Unfix

Buffer Management Operations

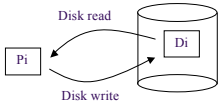
- **Fix**(*pid, flags*)(Also called **Pin**)
  - A fixed page will not be replaced until the page is unfixed.
  - It may involve up to two I/O's, one to write out a dirty page and another to read in the requested page.
- **Unfix**(*pid, flags*)(Also called **Unpin**)
  - decrements by one the counter that indicates the number of transactions that have fixed a page.
  - If this counter is 0 the page is made available for swapping, e.g., it is placed at the tail of the LRU queue.

Buffer Mgmt Operations

- **Touch**(*pid, flags*)
  - sets the *flags* of the page in the buffer table without unfixing the pageBuffer Management Operations
- Possible flags:
  - make the page dirty
  - flush the page
  - fix the page without reading it from disk, etc

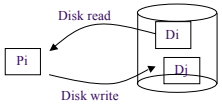
Stable Database

- The *stable database* is the state of the database in stable storage.
- There are two ways to update a data item x in stable storage(*propagation strategies*):
- In-Place Updating: Update x in place (i.e., overwrite x)

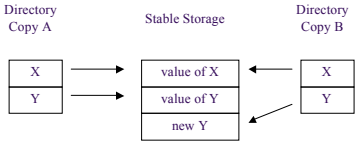


Stable Database

- Shadowing: Write the new value of x in a copy (older versions are called *shadow copy*).



In this case, there must be a directory in stable storage to tell where each item is.



Logging

- A *Log* is a sequence of records which represent all modifications to the database. Log records may describe either *physical* changes or *logical* database operations.
  - A *physical log* contains information about the actual values of data items written by transactions.
    - state before change, *before image*
    - state after change, *after image*
    - transition causing the change
  - A *logical log* represents higher level operations; e.g., insert this key in that index.

Logging

- The order in which updates appear is the same as the order in which they actually occurred. The precise way history is represented in the log depends on the technique followed by the recovery manager.

Log Records

- For the moment, we will assume that a log record may be one of the following types:
  - Start Record
    - [T_i, start]
  - Commit Record
    - [T_i, commit]
  - Abort Record
    - [T_i, abort]

Log Records

- Update Record for physical state logging at page level
  - [T_i, x, b, a]
    - T_i : the id of the transaction that performed a Write operation on x
    - x: the id of data item x
    - b: *before* image of x
    - *a: after* image of x
  - Assuming Strict Executions
    - [T_j, x, b]: T_j wrote into x before T_i
    - [T_i, x, a]

Log Records

- Update Record for physical transition logging on page level
  - [T_i,x, b, d]
  - d is the difference between the before and after images
  - d = before ⊗ after

Logical Logging on the Record Level

- Simply record the operation and its arguments
  [T_i, Op, Inv-op, Arg]
  - Op = {Insert, Delete, Update} [REDO]
  - Inv-op = inverse operation [UNDO]
  - Arg = arguments

=> It is not possible in all models to automatically generate the inverse; e.g., the network model.

Undoing and Redoing Writes

UNDO Rule (*WAL, Write Ahead Logging principle*)

T writes x
T aborts or System crash

– If x was transferred to disk, then we need the *before image* of *x* to *undo* this update.

- Thus, when x is updated by *T*, the DM should store first the *before image* of x in the log on stable storage and then x itself in the stable database.

Undoing and Redoing Writes

REDO Rule

T writes x
T commits
System crash

– If *x* was not transferred to disk, at *restart* time we need the *after image* of *x* to redo *T*'s update.

- Thus, the DM should not commit a transaction *T* until the *after image* of each data item written by *T* is in stable storage.

BM Table for Buffered Log

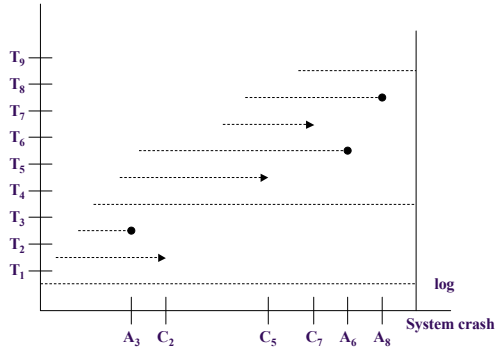
page Id	dirty bit	fix count	block LSN	buffer number
x	0	0	812	0
y	1	1	10	1
z	0	1	123	2

- The Undo rule is:
  - Before the BM replaces a block it should flush all log entries whose LSN is less than or equal to the LSN recorded on this buffer block.

Restarts

- *Restart*: consult the log and for each transaction T_i do the following:
  - **redo** the updates of T_i if there is a commit record of T_i in the log.
  - **Undo** the updates of T_i if there is no such record in log (i.e., T_i had been aborted or it was active when the system crashed).

What should Restart do?



Idempotence of Restarts

- The restart operation may be interrupted because of a failure. Incomplete executions of Restart followed by a completed Restart must have the same effect as just one completed Restart.

Garbage Collection

- Recycling space in the log occupied by unnecessary info.
- Garbage Collection Rule:

  The entry [T_i ,x,v] can be removed from the log *iff*

  - T_i has aborted
  - T_i has committed but some other committed transaction wrote into x after T_i did.

    Note that the last committed value of a data item x must be in a log, if undo is possible.
  - [T_i ,x,v] can be removed from the log if v is the last committed value of x and v is the value of x in the stable storage and there are no other entries of x.

Checkpoints

- To Restart, we need to scan the entire log !
  - The Restart operation will be prohibitively slow.
  - The Log file may become very long and may not fit on disk.
- Most of the transactions that need to be redone have already written their updates to stable database (why?).
  - Thus, most of the Restart operations are unnecessarily performed.
- The amount of work Restart has to do after a system failure can be reduced by *check pointing* the updates that have been performed up to a certain time.

Restart with Checkpointing

- Restart may proceed as before, i.e.:
  - redo updates of transactions that have been committed,undo updates of transactions that have not been committed.

    Notice: The undo procedure may require reading log records written before the most recent checkpoint point (why?).
- In addition, the following scenario for a transaction *T* is possible:
  - *T* was active when the system crashed. *T*, but did not perform any Write operation since the last checkpoint; i.e., there is no record for *T* in the log after the last checkpoint.

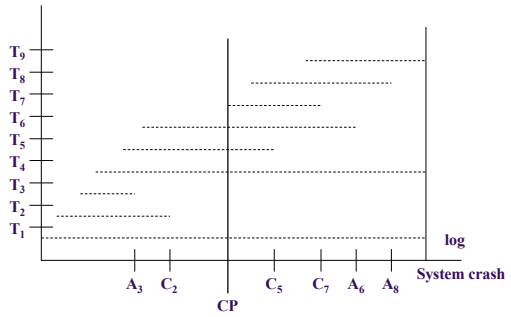
Restart with Checkpointing

=> How can Restart identify such transactions without reading the entire log?

- Checkpoint Record
  - The checkpoint record must include a list of transactions that were active at checkpoint time.

    [checkpoint, Ac]
- A side effect: the start record of a transaction is not needed anymore.

Example: Restart with Checkpoint



Transaction-Oriented Checkpoint

- Force discipline avoids REDO
  - during commit all the updates of the committed transaction are propagated to the stable database.
- Commit can be seen as a checkpoint

Problem:

- Hot spots need to be propagated every time a transaction commits
  - overhead on normal processing.

Transaction Consistent Checkpoint
(Commit Consistent Checkpoint)

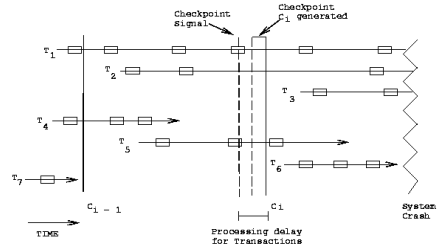
- Stop accepting new *transactions* and wait until all transactions terminate (commit or abort).
- Flush all dirty buffer blocks to stable storage.
- Force-write a <checkpoint> record to log.
- Resume normal execution.

- On Restart, repeat the same steps as before but now process updates of those transactions that appear after the most recent checkpoint record.

Drawback ?

Action Consistent Checkpoint
(Cache Consistent Checkpoint)

- Stop accepting new *operations* (active transactions are blocked).
- Flush all dirty buffer blocks to disk.
- Force-write a <checkpoint> record to the log file.
- Resume normal operation.



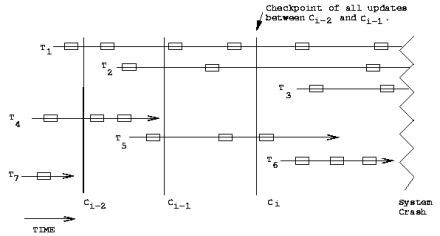
Action Consistent Checkpoint
(Cache Consistent Checkpoint)

Consequences:

- All updates of transactions committed before the checkpoint are now in the stable database (... good).
- All undo updates of transactions aborted before the checkpoint are now in the stable database (... good).
- All updates of transactions that were active at the checkpoint are now in stable database (... bad).

Fuzzy Checkpointing

We can further reduce the delay caused by the checkpoint procedure by flushing those dirty buffers that have not been flushed since before the previous checkpoint.



Fuzzy Checkpointing

- The hope is that the BM might have already flushed the buffers that were dirty before the previous checkpoint. So, the checkpoint will not have much flushing to do.
- We are sure that, at any time, no committed (or aborted) updates recorded before the *penultimate* checkpoint (i.e., the next to last) will have to be redone (or undone).