Chapter 15: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Recovery With Concurrent Transactions
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Advanced Recovery Techniques

Failure Classification

- Transaction failure :
  - Logical errors: transaction cannot complete due to some internal error condition
  - System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash: a power failure or other hardware or software failure causes the system to crash. It is assumed that non-volatile storage contents are not corrupted.
- Disk failure: a head crash or similar failure destroys all or part of disk storage

Storage Structure

- Volatile storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatile storage:
  - survives system crashes
  - examples: disk, tape
- Stable storage:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks; copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can result in inconsistent copies
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Stable-Storage Implementation (Cont.)

- Protecting storage media from failure during data transfer (cont.):
  - Copies of a block may differ due to failure during output operation. To recover from failure:
    1. First find inconsistent blocks:
       (a) *Expensive solution:* Compare the two copies of every disk block.
       (b) *Better solution:* Record in-progress disk writes on non-volatile storage. Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
    2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

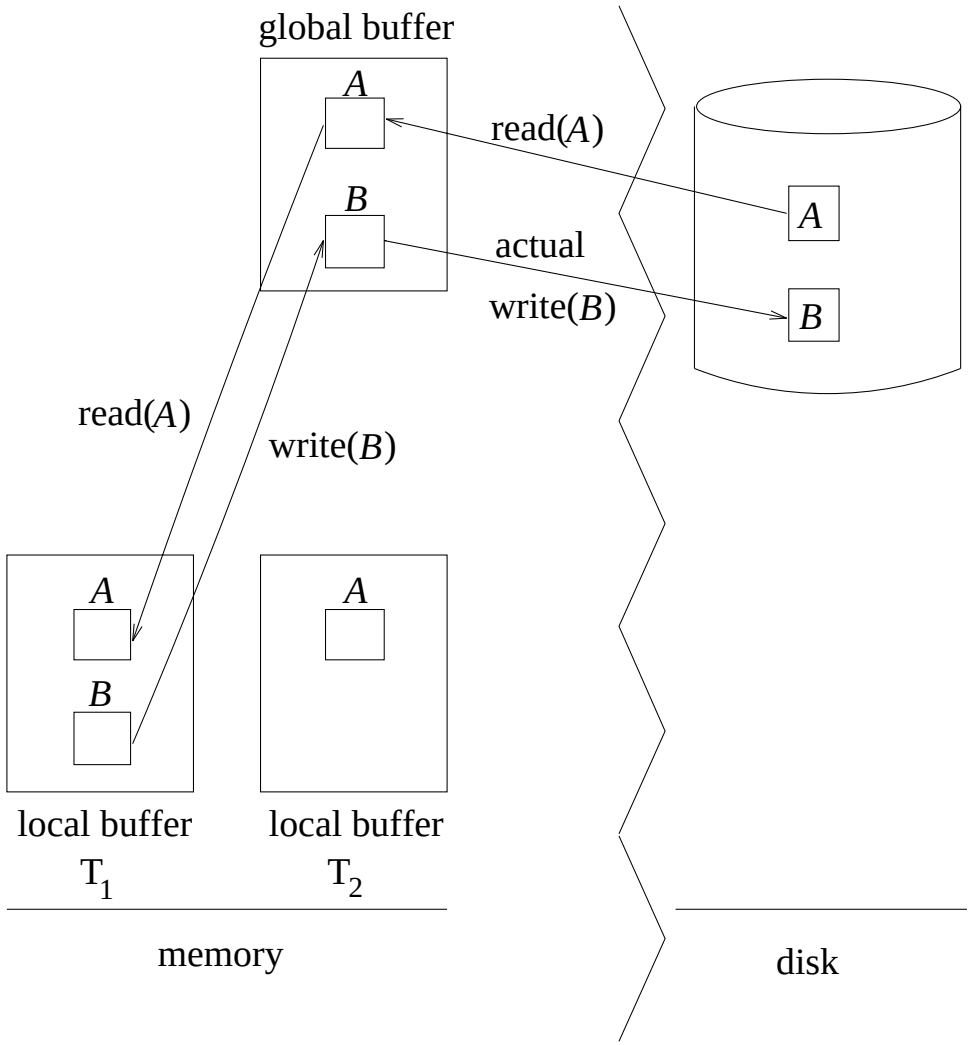
Data Access

- *Physical blocks* are those blocks residing on the disk. *Buffer blocks* are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(B) transfers the physical block B to main memory.
  - **output**(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept. T_i's local copy of a data item X is called x_i.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item X in the buffer block.
  - both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.
- Transactions perform **read**(X) while accessing X for the first time; all subsequent accesses are to the local copy. After last access, transaction executes **write**(X).
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access



Recovery and Atomicity

- Consider transaction T_i that transfers \$50 from account A to account B; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.

Log-Based Recovery

- A *log* is kept on stable storage. The log is a sequence of *log records*, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a $< T_i$ **start** $>$ log record
- *Before* T_i executes **write**(X), a log record $< T_i,\ X,\ V_1,\ V_2 >$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
- When T_i finishes it last statement, the log record $< T_i$ **commit** $>$ is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Deferred Database Modification

- This scheme ensures atomicity despite failures by recording all modifications to log, but deferring all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $< T_i$ **start** $>$ record to log.
- A **write**(X) operation results in a log record $< T_i,\ X,\ V >$ being written, where V is the new value for X. The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $< T_i$ **commit** $>$ is written to the log
- Finally, log records are used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be *redone* if and only if both $< T_i$ **start** $>$ and $< T_i$ **commit** $>$ are there in the log.
- Redoing a transaction T_i (**redo**(T_i)) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken
- example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read**(A)
$A := A - 50$
write(A)
read(B)
$B := B + 50$
write(B)

T_1: **read**(C)
$C := C - 100$
write(C)

Deferred Database Modification (Cont.)

- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ **start**$>$	$<T_0$ **start**$>$	$<T_0$ **start**$>$
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ **commit**$>$	$<T_0$ **commit**$>$
	$<T_1$ **start**$>$	$<T_1$ **start**$>$
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ **commit**$>$

- If log on stable storage at time of crash is as in case:
  - (a) No redo actions need to be taken
  - (b) **redo**(T_0) must be performed since $<T_0$ **commit**$>$ is present
  - (c) **redo**(T_0) must be performed followed by **redo**(T_1) since $<T_0$ **commit**$>$ and $<T_1$ **commit**$>$ are present

Immediate Database Modification

- This scheme allows database updates of an uncommitted transaction to be made as the writes are issued; since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	Write	Output
$<T_0$ **start**$>$		
$<T_0,\ A,\ 1000,\ 950>$		
$<T_0,\ B,\ 2000,\ 2050>$		
	$A = 950$	
	$B = 2050$	
$<T_0$ **commit**$>$		
$<T_1$ **start**$>$		
$<T_1,\ C,\ 700,\ 600>$		
	$C = 600$	
		B_B, B_C
$<T_1$ **commit**$>$		
		B_A

- Note: B_X denotes block containing X.

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one :
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record $< T_i$ **start**$>$, but does not contain the record $< T_i$ **commit**$>$.
  - Transaction T_i needs to be redone if the log contains both the record $< T_i$ **start**$>$ and the record $< T_i$ **commit**$>$.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ **start**$>$	$<T_0$ **start**$>$	$<T_0$ **start**$>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ **commit**$>$	$<T_0$ **commit**$>$
	$<T_1$ **start**$>$	$<T_1$ **start**$>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ **commit**$>$

Recovery actions in each case above are:

(a) **undo**(T_0): B is restored to 2000 and A to 1000.

(b) **undo**(T_1) and **redo**(T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

(c) **redo**(T_0) and **redo**(T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

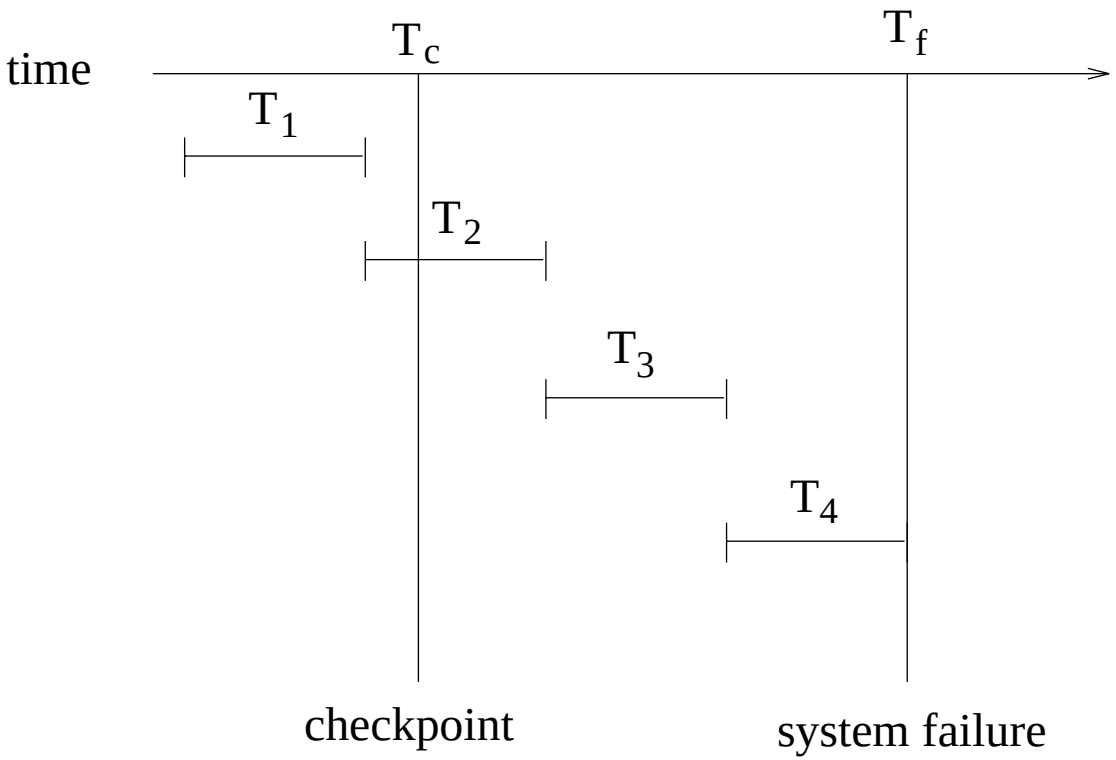
Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing *checkpointing*
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record $<$**checkpoint**$>$ onto stable storage.

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
- Scan backwards from end of log to find the most recent $<$**checkpoint**$>$ record
- Continue scanning backwards till a record $<T_i$ **start**$>$ is found.
- Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
- For all transactions (starting from T_i or later) with no $<T_i$ **commit**$>$, execute **undo**(T_i). (Done only in case of immediate modification.)
- Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ **commit**$>$, execute **redo**(T_i).

Example of Checkpoints

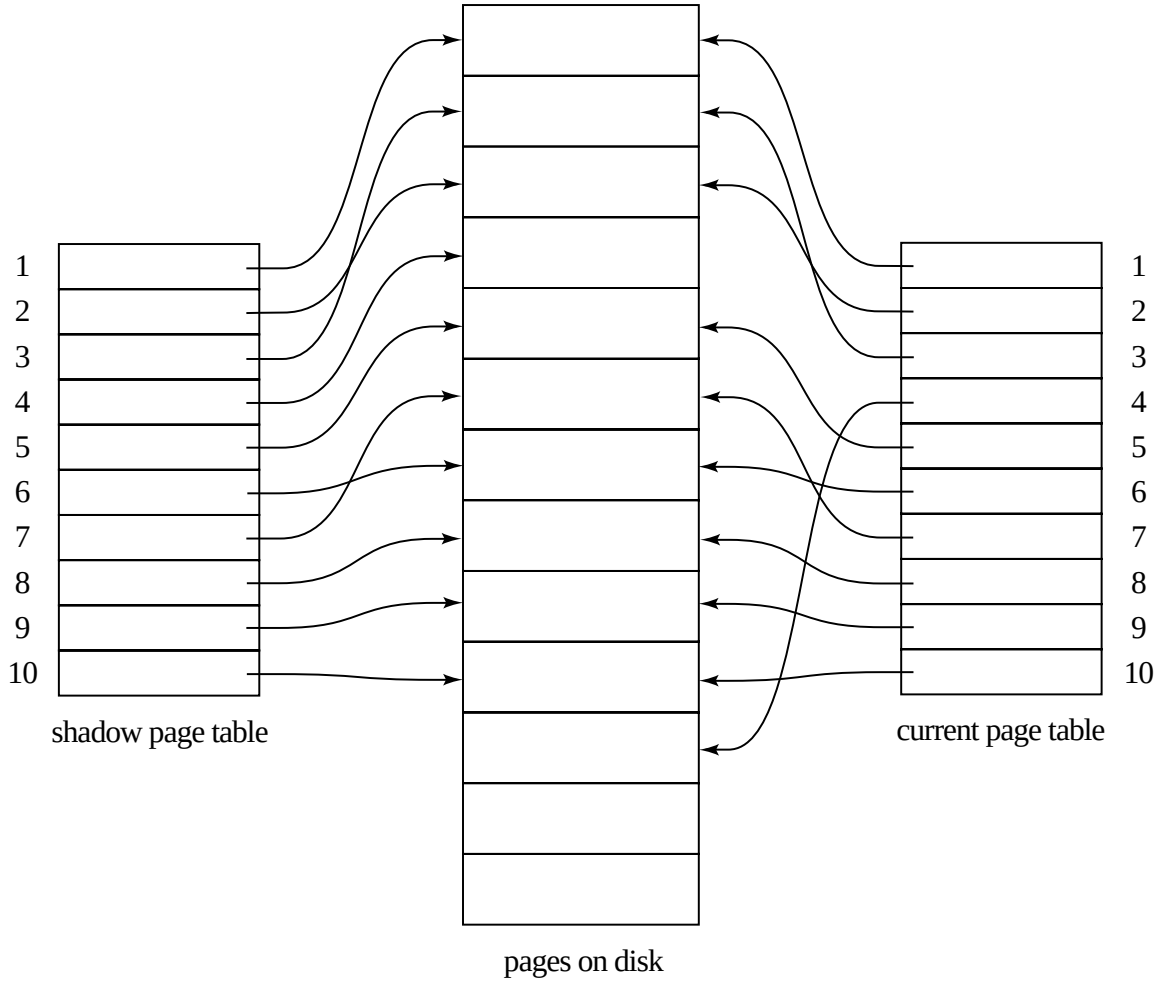


- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

Shadow Paging

- Alternative to log-based recovery
- Idea: maintain *two* page tables during the lifetime of a transaction – the *current* page table, and the *shadow* page table
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered. Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time, a copy of this page is made onto an unused page. The current page table is then made to point to the copy, and the update is performed on the copy

Example of Shadow Paging



Shadow and current page tables after write to page 4

Shadow Paging (Cont.)

- To commit a transaction:
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page the new shadow page table
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Shadow Paging (Cont.)

- Advantages of shadow-paging over log-based schemes – no overhead of writing log records; recovery is trivial
- Disadvantages :
  - Commit overhead is high (many pages need to be flushed)
  - Data gets fragmented (related pages get separated)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected and put into the list of unused pages
  - Hard to extend algorithm to allow transactions to run concurrently

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume concurrency control using strict two-phase locking; ie. the updates of uncommitted transactions should not be visible to other transactions
- Logging is done as described earlier. Log records of different transactions may be interspersed in the log.
- The checkpointing technique and actions taken on recovery have to be changed, since several transactions may be active when a checkpoint is performed.

Recovery With Concurrent Transactions (Cont.)

- Checkpoints are performed as before, except that the checkpoint log record is now of the form $<$**checkpoint** $L>$, where L is the list of transactions active at the time of the checkpoint.
- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first $<$**checkpoint** $L>$ record is found. For each record found during the scan:
     - if the record is $<T_i$ **commit**$>$, add T_i to *redo-list*
     - if the record is $<T_i$ **start**$>$, then if T_i is not in *redo-list*, add T_i to *undo-list*
  3. For every T_i in L, if T_i is not in *redo-list*, add T_i to *undo-list*

Recovery With Concurrent Transactions (Cont.)

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  4. Scan log backwards from most recent record, stopping when $< T_i \ \mathbf{start} >$ records have been encountered for every T_i in *undo-list*.
     During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  5. Locate the most recent <**checkpoint** *L*> record.
  6. Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
     During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*.

Example of Recovery

Go over the steps of the recovery algorithm on the following log:

$< T_0$ **start**$>$
$<T_0,\ A,\ 0,\ 10>$
$<T_0$ **commit**$>$
$< T_1$ **start**$>$ /* Scan in Step 4 stops here*/
$<T_1,\ B,\ 0,\ 10>$
$< T_2$ **start**$>$
$<T_2,\ C,\ 0,\ 10>$
$<T_2,\ C,\ 10,\ 20>$
$<$**checkpoint** $\{T_1, T_2\}>$
$< T_3$ **start**$>$
$<T_3,\ A,\ 10,\ 20>$
$<T_3,\ D,\ 0,\ 10>$
$<T_3$ **commit**$>$

Log Record Buffering

- Log record buffering: log records are buffered in main memory, instead of of being output directly to stable storage. Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state after the log record $<T_i$ **commit**$>$ has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. (This rule is called the **write-ahead logging** or **WAL** rule.)
- As a result of the write-ahead logging rule, if a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first.

Buffer Management (Cont.)

- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Our checkpointing algorithm requires that no updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed. (Such locks held for short duration are called **latches**.)
  - Before a block is output to disk, the system acquires an exclusive lock on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

Buffer Management (Cont.)

Database buffers are generally implemented in virtual memory in spite of some drawbacks:

- When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
- When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from disk and written to another location on disk, resulting in extra I/O! (Known as *dual paging* problem.)
- Ideally when swapping out a database buffer page, operating system should pass control to database, which in turn outputs page to database instead of to swap space (making sure to output log records first)
  - Dual paging can thus be avoided, but common operating systems do not support such functionality.

Failure with Loss of Nonvolatile Storage

- Periodically **dump** the entire content of the database to stable storage
- No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
  - Output all log records currently residing in main memory onto stable storage.
  - Output all buffer blocks onto the disk.
  - Copy the contents of the database to stable storage.
  - Output a record $<$**dump**$>$ to log on stable storage.
- To recover from disk failure, restore database from most recent dump. Then log is consulted and all transactions that committed since the dump are redone.
- Can be extended to allow transactions to be active during dump; known as *fuzzy* or *online* dump.

Advanced Recovery Techniques

- Support high-concurrency locking techniques, such as those used for B^+-tree concurrency control
- Operations like B^+-tree insertions and deletions release locks early. They cannot be undone by restoring old values (*physical undo*), since once a lock is released, other transactions may have updated the B^+-tree.
- Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as *logical undo*).
- For such operations, undo log records should contain the undo operation to be executed; called *logical undo logging*, in contrast to *physical undo logging*.
- Redo information is logged *physically* (that is, new value for each write) even for such operations.

Advanced Recovery Techniques (Cont.)

Operation logging is done as follows:

- When operation starts, log $< T_i,\ O_j$, **operation-begin**$>$. Here O_j is a unique identifier of the operation instance.
- While operation is executing, normal log records with physical redo and physical undo information are logged.
- When operation completes, $<T_i,\ O_j$, **operation-end**, $U>$ is logged, where U contains information needed to perform a logical undo information.
- If crash/rollback occurs before operation completes, the **operation-end** log record is not found, and the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes, the **operation-end** log record is found, and logical undo is performed using U; the physical undo information for the operation is ignored.

Advanced Recovery Techniques (Cont.)

Rollback of transaction T_i is done as follows:

- Scan the log backwards
  1. If a log record $< T_i,\ X,\ V_1,\ V_2 >$ is found, perform the undo and log a special *redo-only* record $< T_i, X, V_1 >$.
  2. If a $< T_i,\ O_j$, **operation-end**, $U >$ record is found
     - Rollback the operation logically using the undo information U. Updates performed during roll back are logged just like during normal operation execution.
     - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record $< T_i, O_j$, **operation-abort**$>$.
     - Skip all preceding log records for T_i until the record $< T_i, O_j$, **operation-begin**$>$ is found

Advanced Recovery Techniques (Cont.)

- Scan the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a $< T_i, O_j,$ **operation-abort**$>$ record is found, skip all preceding log records for T_i until the record $< T_i, O_j,$ **operation-begin**$>$ is found.
- Stop the scan when the record $< T_i,$ **start**$>$ is found
- Add a $< T_i,$ **abort**$>$ record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Advanced Recovery Techniques (Cont.)

The following actions are taken when recovering from system crash

1. *Repeat history* by physically redoing all updates of *all* transactions, scanning log forward from last $<$ **checkpoint** $L>$ record
   - *undo-list* is set to L initially
   - Whenever $<T_i$ **start**$>$ is found T_i is added to *undo-list*
   - Whenever $<T_i$ **commit**$>$ or $<T_i$ **abort**$>$ is found, T_i is deleted from *undo-list*

This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now *undo-list* contains transactions that are incomplete, that is, have neither committed nor been fully rolled back.

Advanced Recovery Techniques (Cont.)

2. Scan log backwards, performing undo on log records of transactions found in *undo-list*. Transactions are rolled back as described earlier.
   - When $<T_i$ **start**$>$ is found for a transaction T_i in *undo-list*, write a $<T_i$ **abort**$>$ log record.
   - Stop scan when $<T_i$ **start**$>$ records have been found for all T_i in *undo-list*

This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

Advanced Recovery Techniques (Cont.)

- Checkpointing is done as follows:

  1. Output all log records in memory to stable storage
  2. Output to disk all modified buffer blocks
  3. Output to log on stable storage a $<$**checkpoint** $L>$ record.

  Transactions are not allowed to perform any actions while checkpointing is in progress.

- Fuzzy checkpointing allows transactions to progress while the most time consuming parts of checkpointing are in progress

Advanced Recovery Techniques (Cont.)

- Fuzzy checkpointing is done as follows:
  1. Write a $<$**checkpoint** $L>$ log record and force log to stable storage
  2. Note list M of modified buffer blocks
  3. Now permit transactions to proceed with their actions
  4. Output to disk all modified buffer blocks in list M
     - blocks should not be updated while being output, and
     - all log records pertaining to a block must be output before the block is output
  5. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk
- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**.
- Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.