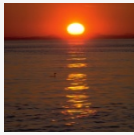
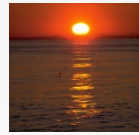
Chapter 17: Recovery System

Database System Concepts



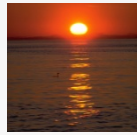
Chapter 17: Recovery System

- 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
- ARIES Recovery Algorithm



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.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

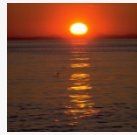


Catastrophic failure

Catastrophic failure is a complete, sudden, often unexpected breakdown in a machine, electronic system, computer or network. Such a breakdown may occur as a result of a hardware event such as a disk drive crash, memory chip failure or surge on the power line. Catastrophic failure can also be caused by software conflicts or malware. Sometimes a single component in a critical location fails, resulting in downtime for the entire system.

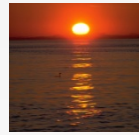
The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is a database backup, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations.



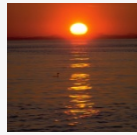
Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability



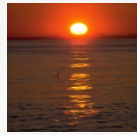
Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media



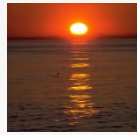
Data Access

- **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.

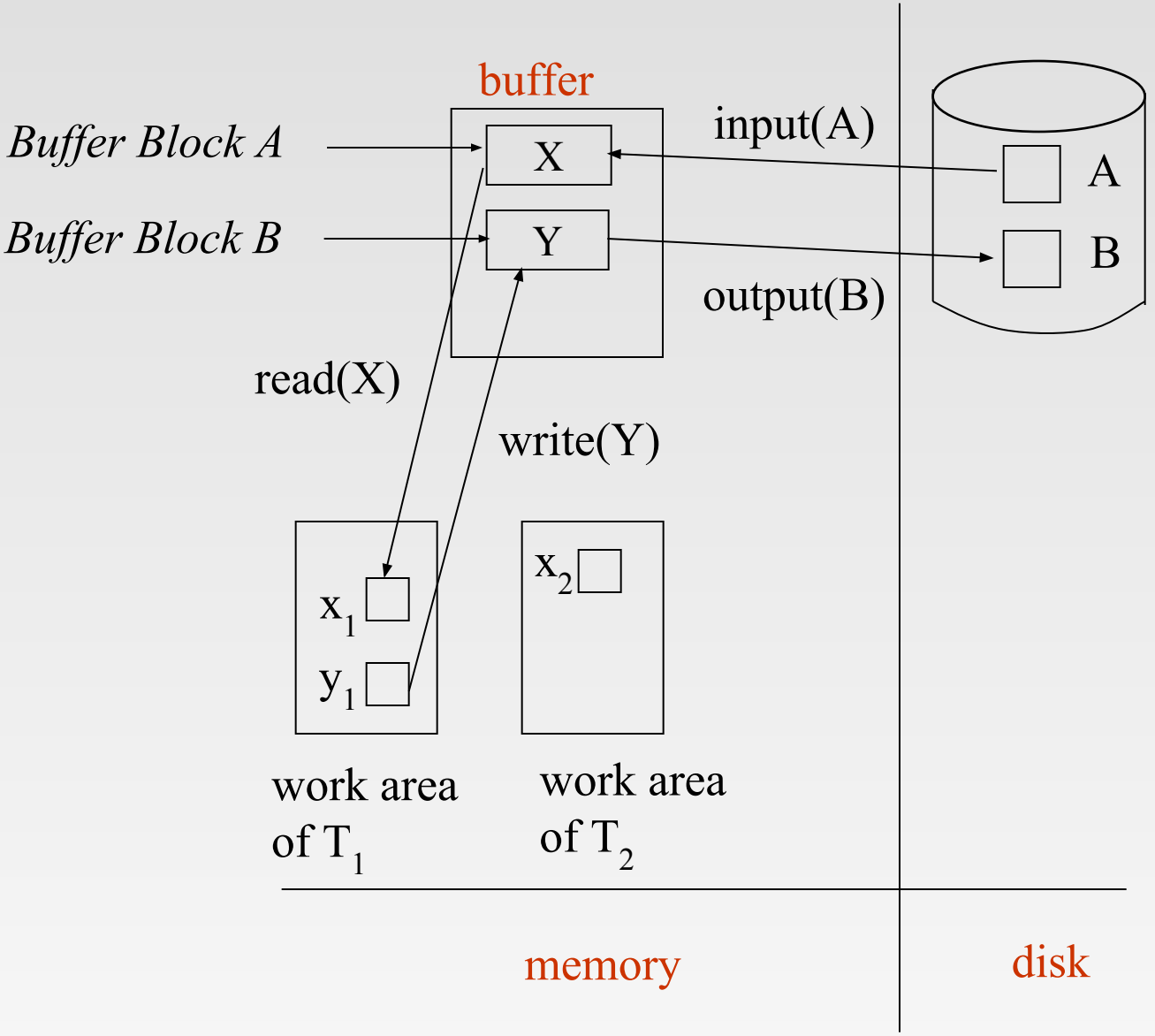


Data Access (Cont.)

- 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.

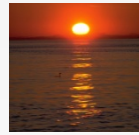


Example of Data Access



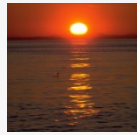
Recovery and Atomicity (Cont.)

- 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.



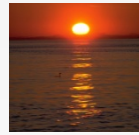
Log-Based Recovery

- 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.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- 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)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification



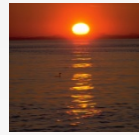
Undo and Redo Operations

- **Undo and Redo of Transactions**
  - **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
    - Each time a data item X is restored to its old value V a special log record $<T_i, X, V>$ is written out
    - When undo of a transaction is complete, a log record $<T_i$ **abort**$>$ is written out.
  - **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
    - No logging is done in this case



Deferred Database Modification

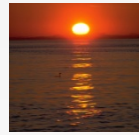
- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- 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, the log records are read and used to actually execute the previously deferred writes.



Deferred Database Modification (Cont.)

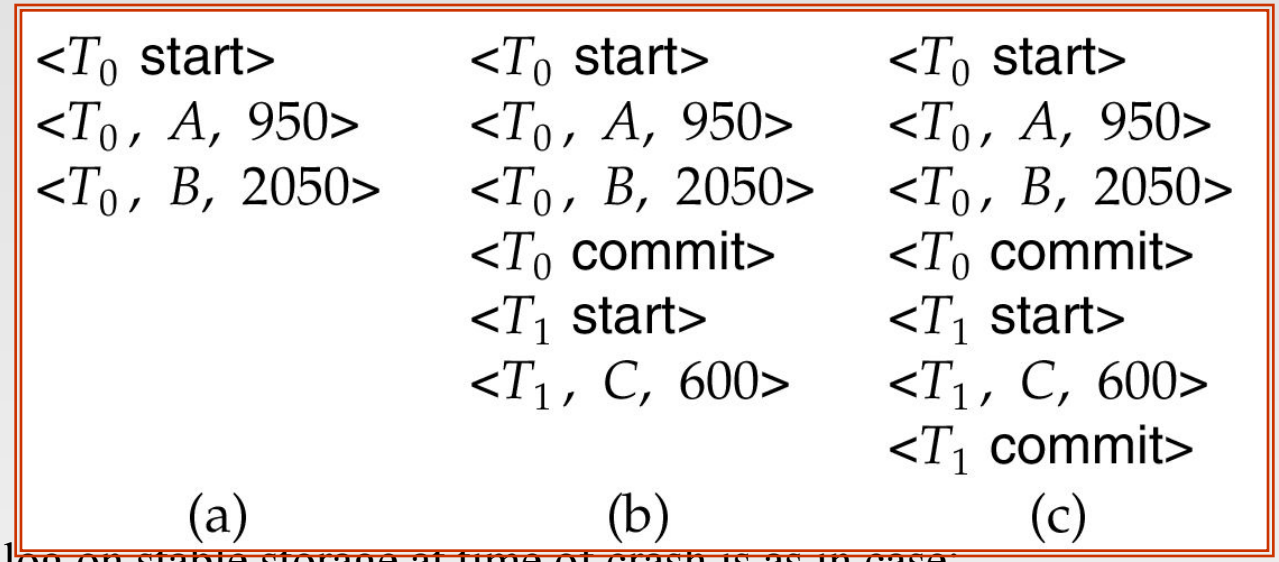
- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction 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)	T_1 : **read** (C)
A: - A - 50	C:- C- 100
Write (A)	**write** (C)
read (B)	
B:- B + 50	
write (B)	



Deferred Database Modification (Cont.)

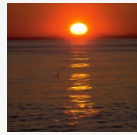
- Below we show the log as it appears at three instances of time.



- 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

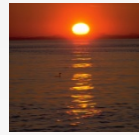
- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- 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

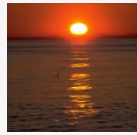
Log	Write	Output
$<T_0$ **start**$>$		
$<T_0$, A, 1000, 950$>$		
T_o, B, 2000, 2050		
	$A = 950$ $B = 2050$	
$<T_0$ **commit**$>$		
$<T_1$ **start**$>$		
$<T_1$, C, 700, 600$>$		
	x_1 $C = 600$	
		B_B, B_C
$<T_1$ **commit**$>$		
		B_A

- 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
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- 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.

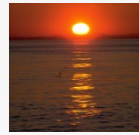
(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

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

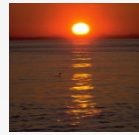
(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

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



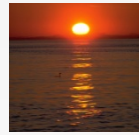
Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already Committed.
  3. output their updates to the database.
- Streamline recovery procedure by periodically performing **check pointing**
  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.

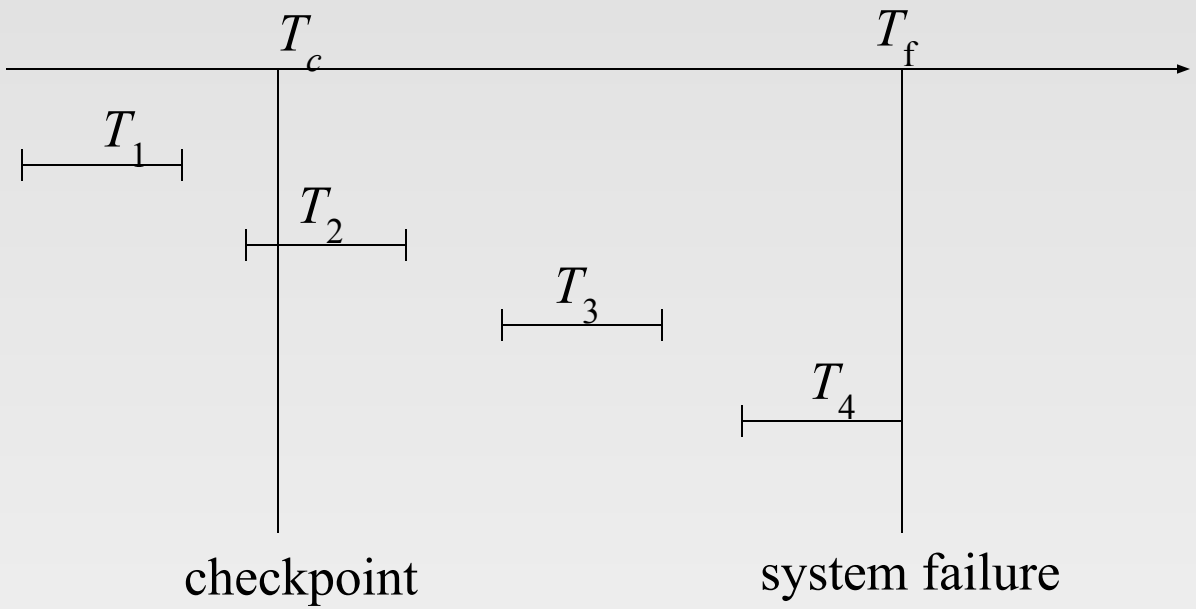


Checkpoints (Cont.)

- 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.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record <T_i **start**> is found.
  3. 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.
  4. 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.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo**(T_i).



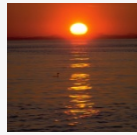
Example of Checkpoints



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

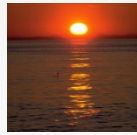
Recovery With Concurrent Transactions

- 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;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- 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.



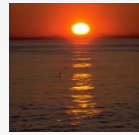
Recovery With Concurrent Transactions (Cont.)

- 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
  - We assume no updates are in progress while the checkpoint is carried out (will relax this later)
- 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 backward 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.)

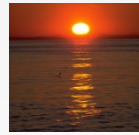
- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when <T_i **start**> records have been encountered for every T_i in *undo-list*.
     - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  2. Locate the most recent <**checkpoint** *L*> record.
  3. 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:

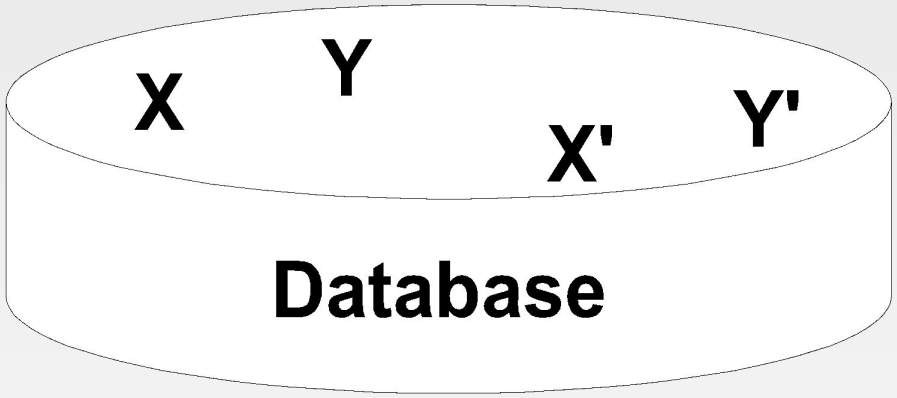
  <T_0 **start**>
  <T_0, A, 0, 10>
  <T_0 **commit**>
  <T_1 **start**> /* Scan at step 1 comes up to here */
  <T_1, B, 0, 10>
  <T_2 **start**>
  <T_2, C, 0, 10>
  <T_2, C, 10, 20>
  <checkpoint $\{T_1, T_2\}$>
  <T_3 **start**>
  <T_3, A, 10, 20>
  <T_3, D, 0, 10>
  <T_3 **commit**>



Database Recovery

Shadow Paging

- The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.



X and Y: Shadow copies of data items
X' and Y': Current copies of data items

Database Recovery

Shadow Paging

- To manage access of data items by concurrent transactions two directories (current and shadow) are used.
  - The directory arrangement is illustrated below. Here a page is a data item. (Figure 23.4)

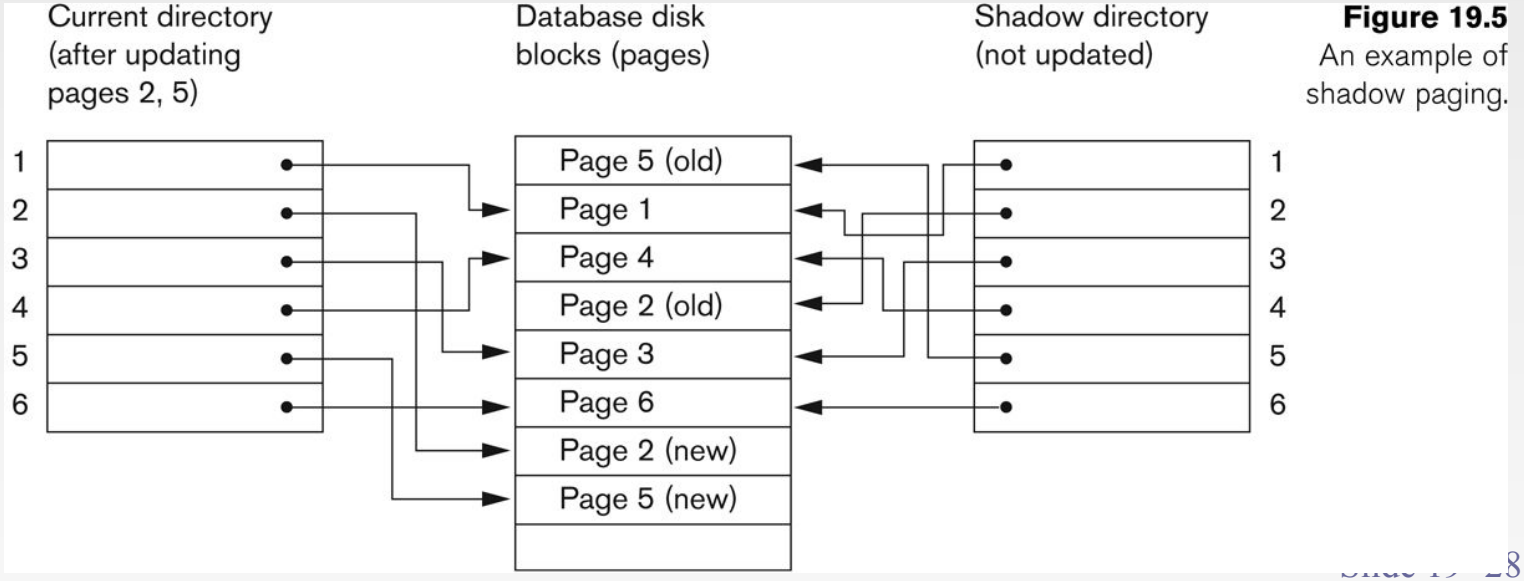
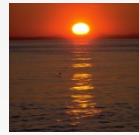


Figure 19.5 An example of shadow paging.

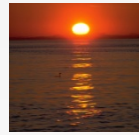
Shadow Paging

- This recovery scheme does not require the use of a log in a single-user environment.
- In a multiuser environment, a log may be needed for the concurrency control method.
- Shadow paging considers the database to be made up of a number of fixed size disk pages (or disk blocks)—say, *n*—*for recovery purposes. A* ***directory with n*** entries is constructed, where the *ith entry points to the ith database page on disk.*
- The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it.
- When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory. The shadow directory is then saved on disk while the current directory is used by the transaction.



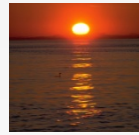
Shadow Paging contd….

- During transaction execution, the shadow directory is *never modified. When a* write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten. Instead, the new page is written* elsewhere—on some previously unused disk block.
- The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.
- Figure 23.4 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.



Shadow Paging contd….

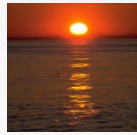
- To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory.
- The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory.
- The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded.
- Committing a transaction corresponds to discarding the previous shadow directory.
- Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.



Shadow Paging contd….

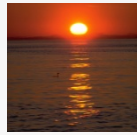
- One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies.
- Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle **garbage collection when a transaction commits.**
- **The old** pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits.
- Another issue is that the operation to migrate between current

  and shadow directories must be implemented as an atomic operation.



Why Concurrency Control is needed:

- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.



Database Recovery

(a)

T_1	T_2	T_3
read_item(A)	read_item(B)	read_item(C)
read_item(D)	write_item(B)	write_item(B)
write_item(D)	read_item(D)	read_item(A)
	write_item(D)	write_item(A)

Figure 19.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

Database Recovery

(b)

		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
**	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

← System crash

Figure 19.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

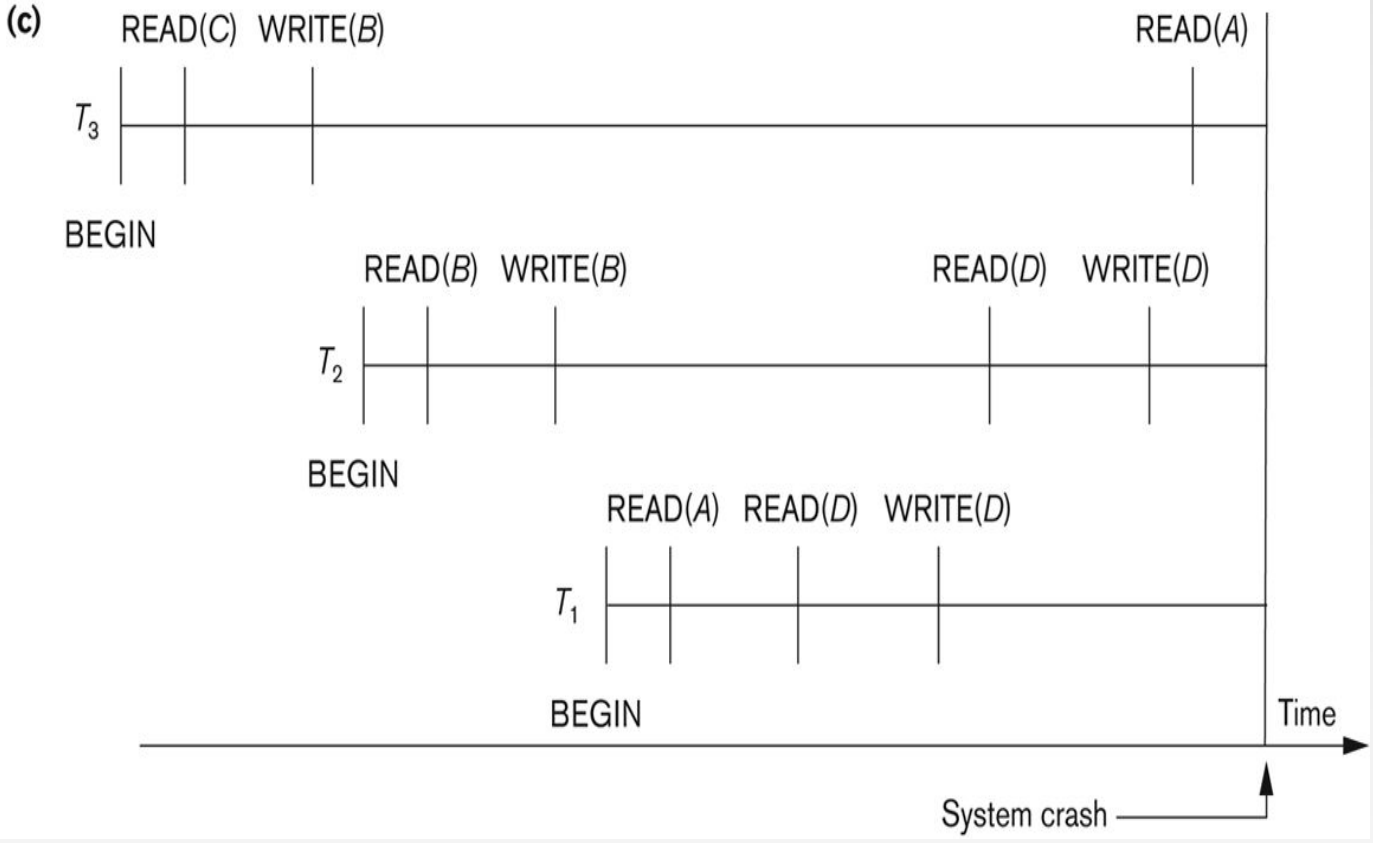
* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3.

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.

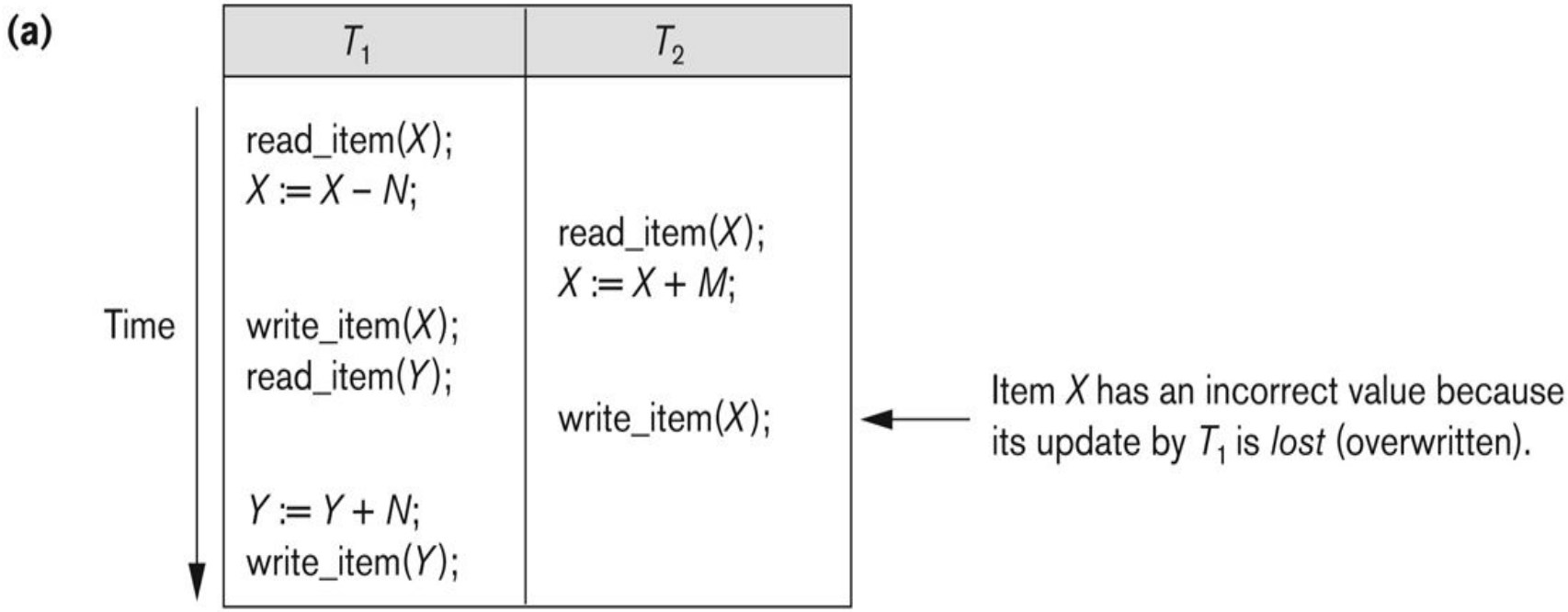
Figure 19.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.



Concurrent execution is uncontrolled: (a) The lost update problem.

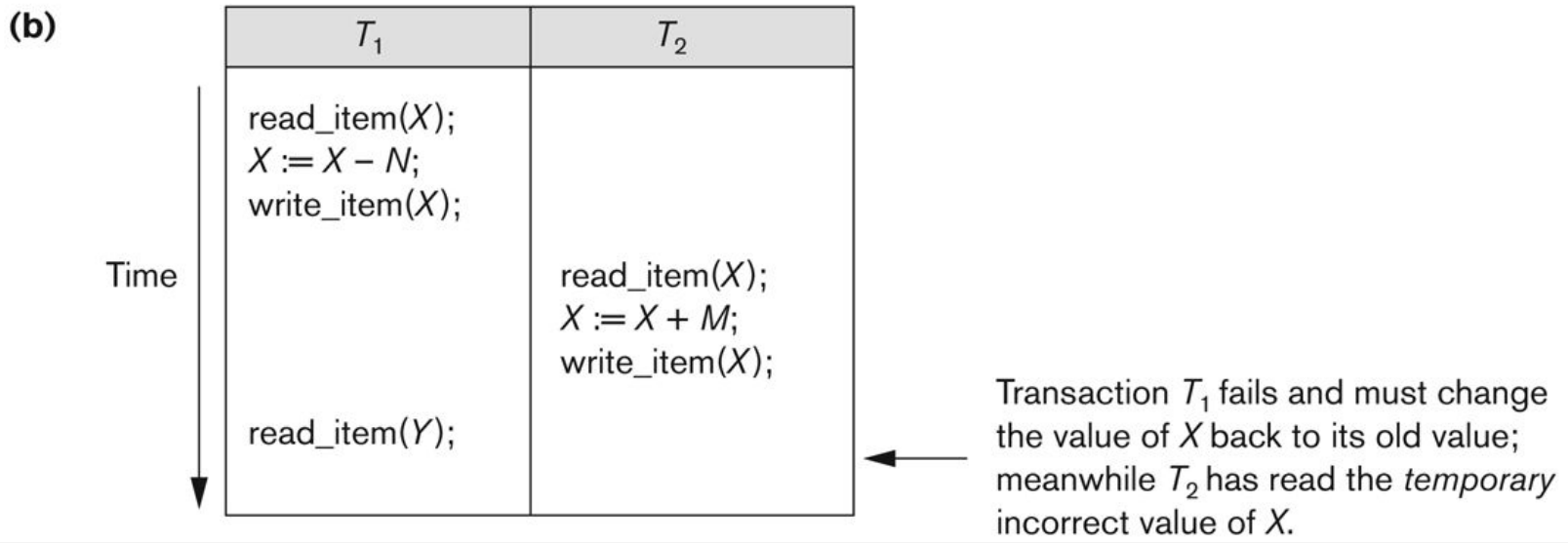
Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Concurrent execution is uncontrolled: (b) The temporary update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



Concurrent execution is uncontrolled: (c) The incorrect summary problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

T_1	T_3
	$sum := 0;$ read_item(A); $sum := sum + A;$ • • •

read_item(X);
$X := X - N;$
write_item(X);	

	read_item(X);
	$sum := sum + X;$
	read_item(Y);
	$sum := sum + Y;$

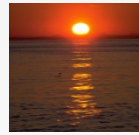
read_item(Y);
$Y := Y + N;$
write_item(Y);	

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Transaction Support in SQL2 (4)

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a transaction which failed.
- **Non repeatable Read**:
  - Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
    4. Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

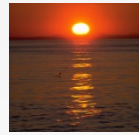


Transaction Support in SQL2 (5)

- **Phantoms:**

New rows being read using the same read with a condition.

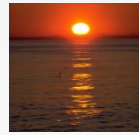
  - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
  - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
  - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.



Transaction Support in SQL2 (7)

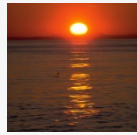
- Possible violation of serializability:

Isolation level	Type of Violation: Dirty read	nonrepeatable read	phantom
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no



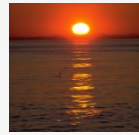
Log Record Buffering

- **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.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.



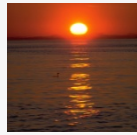
Log Record Buffering (Cont.)

- 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 only when 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
      - Strictly speaking WAL only requires undo information to be output



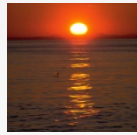
Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- 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
  - (Write ahead logging)
- 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 latch on the block
    - Ensures no update can be in progress on the block



Buffer Management (Cont.)

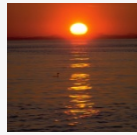
- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs 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.)

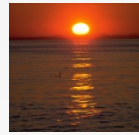
- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    4. Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use

    Dual paging can thus be avoided, but common operating systems do not support such functionality.



Failure with Loss of Nonvolatile Storage

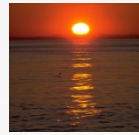
- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing 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.



Advanced Recovery: Checkpointing

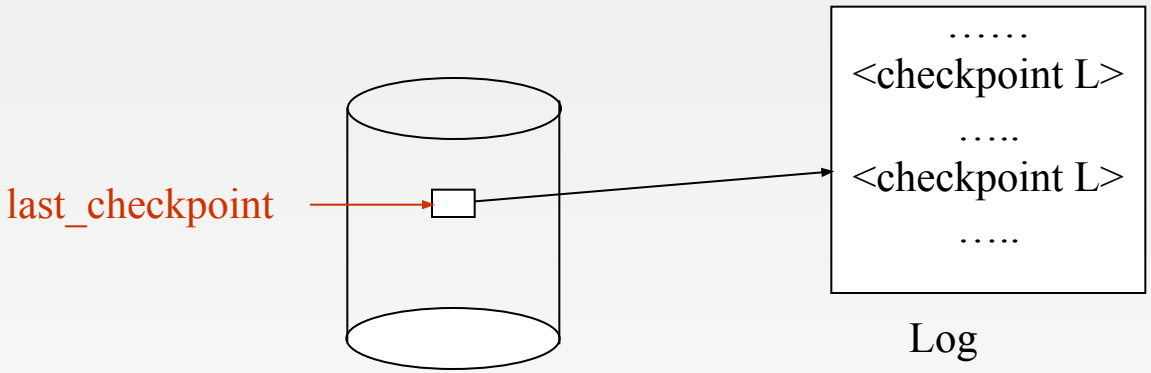
- **Checkpointing** is done as follows:
  1. Output all log records in memory to stable storage
  2. Output to disk all modified buffer blocks
  3. Output to log on stable storage a < **checkpoint** *L*> record.

  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
  - Performed as described on next slide



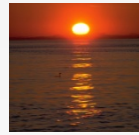
Advanced Recovery: Fuzzy Checkpointing

- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk



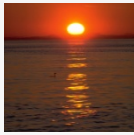
Advanced Rec: Fuzzy Checkpointing (Cont.)

- 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.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



ARIES Recovery Algorithm

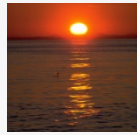
Database System Concepts



Database Recovery

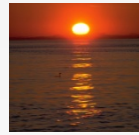
The ARIES Recovery Algorithm (contd.)

- The following steps are performed for recovery
  - **Analysis phase**: Start at the begin_checkpoint record and proceed to the end_checkpoint record. Access transaction table and dirty page table are appended to the end of the log. Note that during this phase some other log records may be written to the log and transaction table may be modified. The analysis phase compiles the set of redo and undo to be performed and ends.
  - **Redo phase**: Starts from the point in the log up to where all dirty pages have been flushed, and move forward to the end of the log. Any change that appears in the dirty page table is redone.
  - **Undo phase**: Starts from the end of the log and proceeds backward while performing appropriate undo. For each undo it writes a compensating record in the log.
- The recovery completes at the end of undo phase.



Example

- Consider the recovery example shown in Figure 23.5. There are three transactions:
- T1, T2, and T3. T1 updates page C, T2 updates pages B and C, and T3 updates page A.



Database Recovery

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure 19.6

An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.