Programme M.Sc. in Computer Science

Course – Advanced DBMS

Module – Database Recovery

BIBEK RANJAN GHOSH

(Assistant Professor, Department of Computer Science)

DATABASE RECOVERY TECHNIQUE

Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution,

1. **Committed,** the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or
2. **Aborted,** that the transaction does not have any effect on the database or any other transactions.

Types of Failures

1. A computer failure (system crash). A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

2. A transaction or system error. Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the transaction during its execution

3. Local errors or exception conditions detected by the transaction. During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition,4 such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.

4. Concurrency control enforcement. The concurrency control method may decide to abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

5. Disk failure. Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.

6. Physical problems and catastrophes. This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

The System Log

- To recover from failures, the system maintains a **log to keep track of all transaction operations that affect the values of database** items, as well as other transaction information that may be needed to permit recovery from failures.
- The log is a sequential, append-only file that is kept on disk, so it s not affected by any type of failure except for disk or catastrophic failure.
- **One (or more) main memory buffers hold the last part of the log file**, so that log entries are first added to the main memory buffer. When the **log buffer is filled, or** when certain other conditions occur, the log buffer **is *appended to the end of the log file on disk.***

Log record

1. **[start_transaction, *T]. Indicates that transaction T has started execution.***
2. **[write_item, *T, X, old_value, new_value]. Indicates that transaction T has** changed the value of database item *X from old_value to new_value.*
3. **[read_item, *T, X]. Indicates that transaction T has read the value of database** item *X.*
4. **[commit, *T]. Indicates that transaction T has completed successfully, and** affirms that its effect can be committed (recorded permanently) to the database.
5. **[abort, *T]. Indicates that transaction T has been aborted.***

Undo :

Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo the effect of these WRITE operations of a transaction *T by tracing*** backward through the log and resetting all items changed by a WRITE operation of *to their old_values.*

Redo

Redo of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can be sure that all these new_values have been written to the actual database on disk from the main memory buffers.

Commit Point of a Transaction

1. A transaction *T reaches its* ***commit point when all its operations that access the d***atabase have been executed successfully *and the effect of all the transaction operations* on the database have been recorded in the log.

- Beyond the commit point, the transaction is said to be **committed, and its effect must be *permanently recorded in*** the database.
- The transaction then writes a **commit record [commit, *T]*** *into the log.*

1. If a system failure occurs,

- we can search back in the log for all transactions *T that* have written a [start_transaction, *T]* *record into the log but have not written their* [commit, *T] record yet;*
- *these transactions may have to be rolled back to undo their effect on the database during the recovery process.*

1. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone from the log records.*

1. It is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer, until they are filled with log entries and then to write them** back to disk only once, rather than writing to disk every time a log entry is added.
2. At the ime of a system crash, only the log entries that have been *written back to disk are* considered in the recovery process because the contents of main memory may be Lost
3. **Hence, *before a transaction reaches its commit point, any portion of the log* that has not been written to the disk yet must now be written to the disk.** This process is called **force-writing the log buffer before committing a transaction.**

Typical strategy for recovery

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash,

- the recovery method restores a past copy of the database that was *backed up to archival storage (tape/other)*
- and reconstructs a more current state by reapplying or *redoing the operations of committed transactions* from the *backed up log, up to the time of failure.*

2. When the database on disk is not physically damaged, and a noncatastrophic Failure has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database.

- undo: transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing its write operations.*
- *redo: some operations* in order to restore a consistent state of the database;

Two main techniques for recovery from noncatastrophic transaction failures

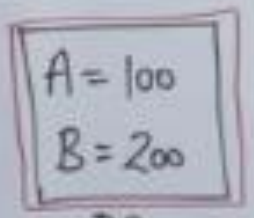
1. **Deferred update techniques**
2. **Immediate update techniques**

1. Deferred update techniques (NO-UNDO/REDO algorithm)

- **Do not physically update the database on disk until *after a transaction*** reaches its commit point; then the updates are recorded in the database.

- Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains.
- Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk
- If a transaction fails before reaching its commit point, it will not have changed database in any way, so UNDO is not needed.
- It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database on disk.

Deferred Database Modification (Log based Recovery)

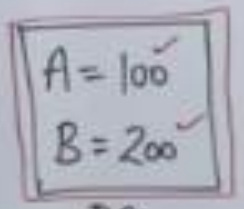


T_1

R(A)
A = A + 100
W(A)
R(B)
B = B + 200
W(B)

$< T_1$, Start $>$	$< T_1$, Start $>$
$< T_1$, A, 200 $>$	$< T_1$, A, 200 $>$
$< T_1$, B, 400 $>$	$< T_1$, B, 400 $>$
$< T_1$, Commit $>$	$< T_1$, Commit $>$
	$< T_2$, Start $>$
	$< T_2$, C, 500 $>$

Deferred Database Modification (Log based Recovery)

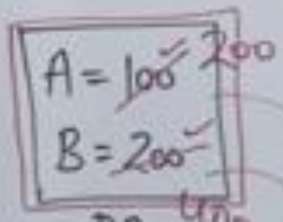


T_1
R(A) 100
A = A + 100
W(A) 200
R(B) 200
B = B + 200
W(B) 400
Commit

Log
$< T_1$, Start $>$
$< T_1, A, 200 >$ New
$< T_1, B, 400 >$ New
$< T_1$, Commit $>$

Log
$< T_1$, Start $>$
$< T_1, A, 200 >$
$< T_1, B, 400 >$
$< T_1$, Commit $>$
$< T_2$, Start $>$
$< T_2, C, 500 >$

Deferred Database Modification (Log based Recovery)

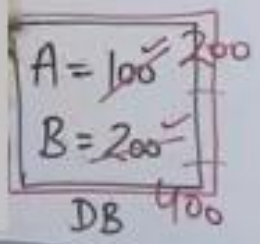


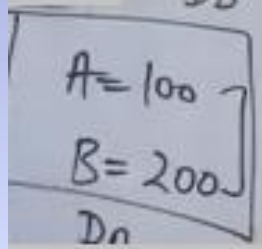
A = 200
B = 400

T_1
R(A) 100
A = A + 100
W(A) 200
R(B) 200
B = B + 200
W(B) 400
Commit

$< T_1$, Start >	$< T_1$, Start >
$< T_1$, A, 200 > (New)	$< T_1$, A, 200 >
$< T_1$, B, 400 > (New)	$< T_1$, B, 400 >
$< T_1$, Commit >	$< T_1$, Commit >
Redo	$< T_2$, Start >
	$< T_2$, C, 500 >

Deferred Database Modification (Log based Recovery)





T_1
R(A) 100
A = A + 100
W(A) 200
R(B) 200
B = B + 200
W(B) 400
* fail

$< T_1, Start >$

$< T_1, A, 200 >$ New

$< T_1, B, 400 >$ New

Redo

$< T_1, Start >$

$< T_1, A, 200 >$

$< T_1, B, 400 >$

$< T_1, Commit >$

$< T_2, Start >$

$< T_2, C, 500 >$

2. Immediate update techniques(UNDO/REDO algorithm)

- The database *may be updated by some operations* of a transaction *before the transaction reaches its commit point.*
- However, these operations must also be recorded in the log *on disk by force-writing before they* are applied to the database on disk, making recovery still possible.
- If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back.
- **B**oth *undo and redo may be required during recovery.*
- A variation of the algorithm where **all updates are required to be recorded in the database on disk *before a transaction commits*** *requires undo* only, so it is known as the **UNDO/NO-REDO algorithm.**

Immediate Database Modification (Log based Recovery)

A = ~~100~~	200
B = ~~200~~	400

DR

T_1

R(A)
A = A + 100
W(A) 200
R(B)
B = B + 200
W(B) 400
Commit

$< T_1, Start >$
$< T_1, A, \overset{Old}{100}, \overset{New}{200} >$
$< T_1, B, 200, 400 >$
$< T_1, Commit >$

Redo

$< T_1, Start >$
$< T_1, A, 1000, 2000 >$
$< T_1, B, 5000, 6000 >$
$< T_1, Commit >$
$< T_2, Start >$
$< T_2, C, 700, 800 >$

Immediate Database Modification (Log based Recovery)

A = ~~100~~	200
B = ~~200~~	400

DB

DB

A = ~~100~~	200
B = ~~200~~	400

T_1

R(A)
A = A + 100
W(A) 200
R(B)
B = B + 200
W(B) 400
⋮
~~Fail~~

$< T_1$, Start $>$	$< T_1$, Start $>$
$< T_1$, A, Old 100, New 200$>$	$< T_1$, A, 1000, 2000
$< T_1$, B, 200, 400$>$	$< T_1$, B, 5000, 6000 $>$
	$< T_1$, Commit $>$
	$< T_2$, Start $>$
	$< T_2$, C, 700, 800 $>$

Immediate Database Modification (Log based Recovery)

A = ~~100~~	200
B = ~~200~~	400

DB

DB

A = ~~100~~ ~~200~~	100
B = ~~200~~	400 200

T_1

R(A)
A = A + 100
W(A) 200
R(B)
B = B + 200
W(B) 400
⋮
~~Fail~~

$< T_1$, Start >	$< T_1$, Start >
$< T_1$, A, (100), 200 >	$< T_1$, A, 1000, 2000 >
$< T_1$, B, (200), 400 >	$< T_1$, B, 5000, 6000 >
Undo	$< T_1$, Commit >
	$< T_2$, Start >
	$< T_2$, C, 700, 800 >

(Old, New)

Caching (Buffering) of Disk Blocks

- Typically, multiple disk pages that include the data items to be updated are **cached into main memory buffers (**an operating system function) **and then updated in memory before being written** back to disk.
- Typically a collection of in-memory buffers, called the **DBMS cache, is** kept under the control of the DBMS for the purpose of holding these buffers.
- A **directory for the cache is used to keep track of which database items are in the** Buffers.
- When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache..
- If not, the item must be located on disk, and the appropriate disk pages are copied into the cache.
- Associated with each buffer in the cache is a **dirty bit, which** can be included in the directory entry, to indicate whether or not the buffer has been modified.

➢**When a page is first read from the database disk into a cache buffer**, a new entry is inserted in the cache directory with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1.*

➢the **pin-unpin bit, is also needed—a page in the cache is pinned (bit value 1 (one)) if it cannot be written back to disk as yet.**

➢Two main strategies can be employed when flushing a modified buffer back to disk

1. **in-place updating, writes the buffer to the *same original d*isk location, thus overwriting the old value of any changed data items on disk.**
2. **shadowing, writes an updated buffer at a different disk location, so multiple** versions of data items can be maintained

1. Write-ahead logging

When in-place updating is used, it is necessary to use a log for recovery. Recovery mechanism must ensure that the **before image** BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the **after image** AFIM in the database on disk. This process is generally known as **write-ahead logging, and is necessary to be able to UNDO the** operation if this is required during recovery

Two types of log entry information included for a write command:

1. A **REDO-type log entry includes the new value** (AFIM) of the item written by the operation since this is needed to *redo the effect of* the operation from the log.
2. The **UNDO-type log entries include the old value (BFIM) of the item since** this is needed to *undo the effect of the operation from the log.*

In an UNDO/REDO algorithm, both types of log entries are combined

When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update *must first be written to disk before the data block itself can be written back to disk from its* main memory buffer

Rules that govern *when a page from the database* can be written to disk from the cache:

no-steal approach.

- If a cache buffer page updated by a transaction *cannot be written to disk* before the transaction commits, the recovery method is called a **no-steal approach. The pin-unpin bit will be used to indicate if a page cannot be** written back to disk.
- The *no-steal rule means that UNDO will never be needed during recovery, since a committed* transaction will not have any of its updates on disk before it commits.
- The deferred update (NO-UNDO) recovery scheme follows a *no-steal approach.*

steal approach.

- On the other hand, if the recovery protocol allows writing an updated buffer *before the transaction commits, it is called* ***steal.***

Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The *advantage of steal is that it avoids the need for a very large buffer space to* store all updated pages in memory

Force/no-force approach

- If all pages updated by a transaction are immediately written to disk *before* the transaction commits, it is called a **force approach.**

- **Otherwise, it is called no-force. The *force rule means that REDO will never be needed during recovery,*** since any committed transaction will have all its updates on disk before it is committed.
- The ***advantage of no-force*** *is that an updated* page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk, and possibly to have to read it again from disk.

To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database.

Consider the following **write-ahead logging (WAL) protocol for a recovery algorithm that requires both UNDO and REDO:**

1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log records for the updating transaction—up to this point—have been force-written to disk.

2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force written to disk.

Checkpoints in the System Log and Fuzzy Checkpointing

➢**A checkpoint log entry [checkpoint, *list of active transactions] record is written into the log periodically at that point when the system* writes out to the database on disk all DBMS buffers that have been modified.**

➢As a consequence of this, all transactions that have their [commit, *T] entries in the log* before a [checkpoint] entry do not need to have their WRITE operations *redone in case* of a system crash, since all their updates will be recorded in the database on disk during check pointing.

➢Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.

2. Force-write all main memory buffers that have been modified to disk

3 .Write a [checkpoint] record to the log, and force-write the log to disk.

4. Resume executing transactions

fuzzy checkpointing

- The time needed to force-write all modified memory buffers may delay transaction processing because of step 1.
- To reduce this delay, it is common to use a technique called **fuzzy checkpointing.**
- **In this technique, the system can resume transaction** processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish.
- When step 2 is completed, an [end_checkpoint, ...] record is written in the log with the relevant information collected during checkpointing.
- However, until step 2 is completed, the previous checkpoint record should remain valid.
- To accomplish this, the system maintains a file on disk that contains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log.
- Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

Transaction Rollback and Cascading Rollback

Transaction Rollback

- If a transaction fails after updating the database, but before the transaction commits, it may be necessary to **roll back the transaction.**
- If any data item values have been changed by the transaction and written to the database, they just be restored to their previous values (BFIMs). The undo-type log entries are used to restore the old values of data items that must be rolled back.

Cascading Rollback

If a transaction *T is rolled back, any transaction S that has, in the interim, read the* value of some data item *X written by T must also be rolled back. Similarly, once S is* rolled back, any transaction *R that has read the value of some data item Y written by S must also be rolled back; and so on. This phenomenon is called* ***cascading rollback,*** and can occur when the recovery protocol ensures *recoverable schedules but* does not ensure *strict or cascadeless schedules*

- quite complex and time-consuming.
- Almost all recovery mechanisms are designed so that cascading rollback *is never required*

(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)

(b)

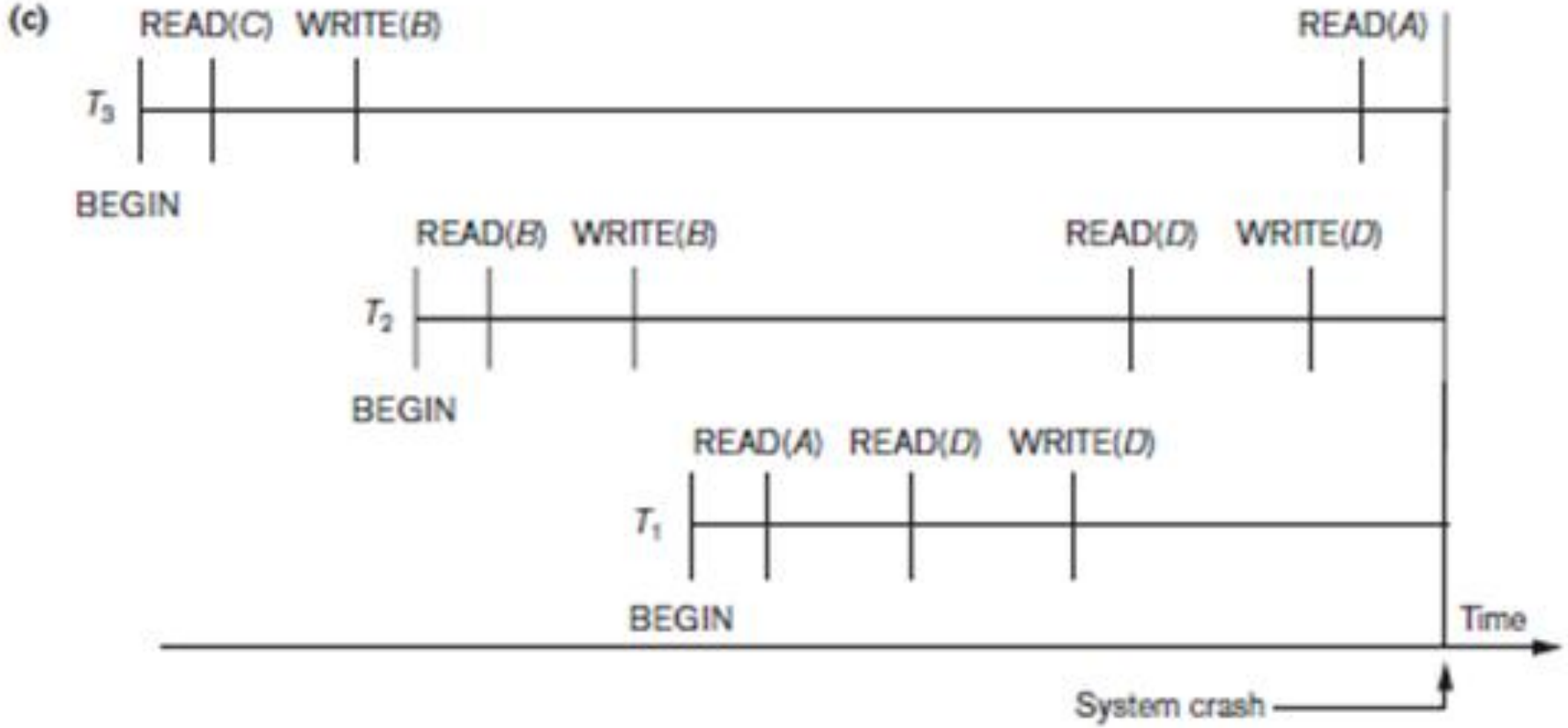
		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
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

← System crash

* 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.

Figure 23.1
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.



Operations before the crash.

NO-UNDO/REDO Recovery Based on Deferred Update

- Defer any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.
- During transaction execution, the updates are recorded only in the log and in the cache buffers.
- After the transaction reaches its commit point and the log is forcewritten to disk, the updates are recorded in the database.
- If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way.
- Therefore, only **REDO type log entries are needed in the log, which include the new value (AFIM) of the** item written by a write operation.
- The **UNDO-type log entries are not needed since** no undoing of operations will be required during recovery
- it cannot be used in practice unless transactions are short and each transaction changes few items.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point.

2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and the log buffer is force-written to disk.*

➢Step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations.

➢REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries during recovery.

- If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk.
- it limits the concurrent execution of transactions because *all write-locked items remain locked until the transaction reaches its commit point.*
- it may require excessive buffer space to hold all updated items until the transactions commit.

The method's main benefit is that transaction operations *never need to be undone, for two reasons:*

1. A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.

2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occu

Recovery Techniques Based on Immediate Update

➢In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately, without any need to wait for the transaction to* reach its commit point.

- it is *not a requirement that every update be* applied immediately to disk; it is just possible that some updates are applied to disk b*efore the transaction commits.*

- *undoing the effect of update operations* by a *failed transaction* is accomplished by rolling back the transaction and undoing the effect of the transaction's write_item operations.
- Therefore, the **UNDO-type log entries, which include the old value (BFIM) of the** item, must be stored in the log.
- Because UNDO can be needed during recovery, these methods follow a **steal strategy for deciding when updated main memory buffers** can be written back to disk

two main categories of immediate update algorithms:

- **UNDO/NO-REDO** If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits, there is never a need to REDO any operations of committed t*ransactions. This is called the **UNDO/NO-REDO recovery algorithm.**
- all updates by a transaction must be recorded on disk *before the transaction commits, so that REDO is never needed.* Hence, this method must utilize the **force strategy for deciding when updated main memory buffers are written back to disk**

➢UNDO/REDO

- If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the **UNDO/REDO recovery algorithm.**
- In this case, the **steal/no-force strategy is applied. This is** also the most complex technique. We will outline an UNDO/REDO recovery algorithm

Procedure RIU_M (UNDO/REDO with checkpoints).
1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the write_item operations of the *active (uncommitted) transactions,* using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the write_item operations of the *committed transactions from the log,* in the order in which they were written into the log, using the REDO procedure defined earlier

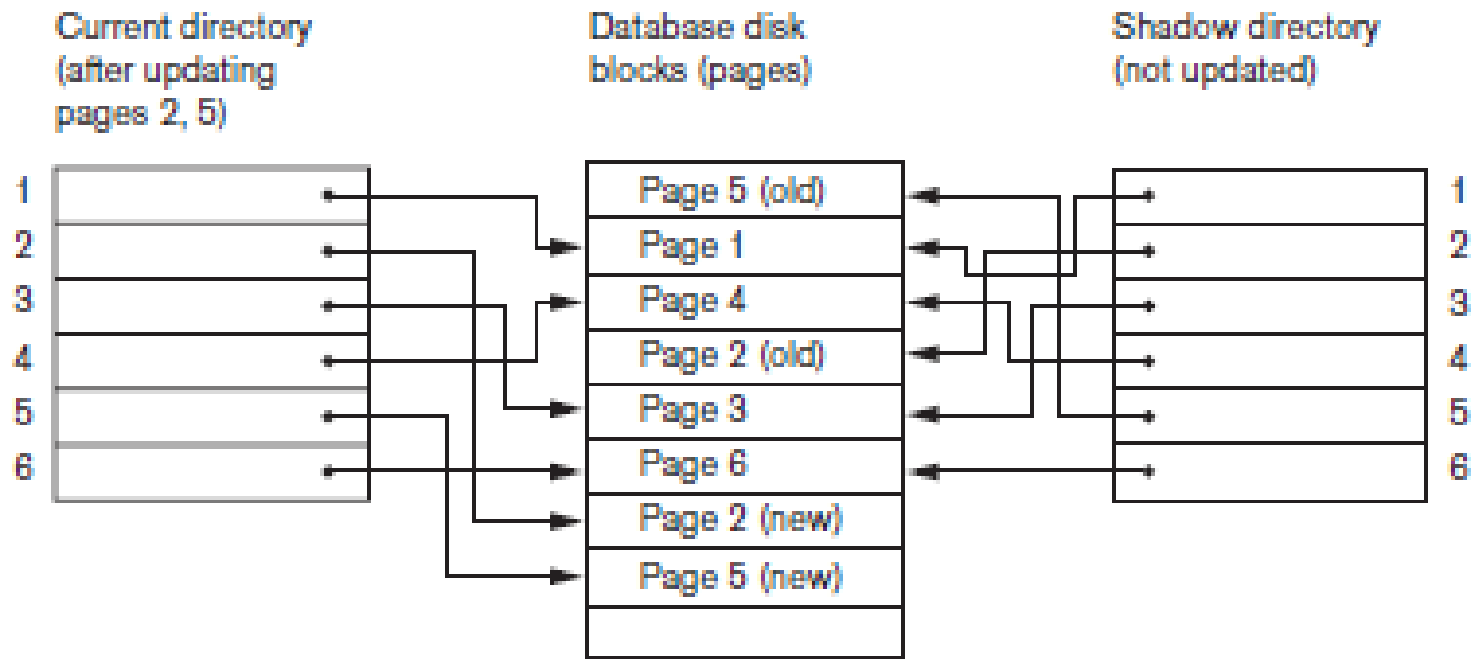
The UNDO procedure is defined as follows:

Procedure UNDO (WRITE_OP). Undoing a write_item operation write_op consists of examining its log entry [write_item, *T, X, old_value, new_value] and setting* the value of item *X in the database to old_value, which is the before image* (BFIM). Undoing a number of write_item operations from one or more transactions from the log must proceed in the *reverse order from the order in which* the operations were written in the log.

Shadow Paging

- Shadow paging considers the database to be made up of a number of fixedsize 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.

Figure 23.4
An example of shadow paging.



[5]The directory is similar to the page table maintained by the operating system for each process.

- 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..
- 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.

ARIES RECOVERY ALGORITHM

- ARIES is a state of the art recovery method, designed to work with a steal, no-force approach.
- Unlike the recovery algorithm described earlier, ARIES
  1. Uses log sequence number (LSN) to identify log records
  2. Dirty page table to avoid unnecessary *redos* during recovery
  3. Transaction Table to record the active transactions at the time of crash

When the recovery manager is invoked after a, crash: restart proceeds in three phases:

- Analysis: Identifies dirty pages in the buffer pool and active transactions at the time of the crash.
- Redo: Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
- Undo: Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

Three main Principles:

Write –Ahead Logging: Any changes to the database object is first recorded in the log and the log has to be written onto a stable storage before the changes are made.

Repeating History: During Redo the algorithm retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was at the time of crash.

Compensation Log Records (CLR): Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated failure.

Prev LSN	Trans ID	Type	Page ID	Length	Offset	Before-Image	After -Image

Common to all records

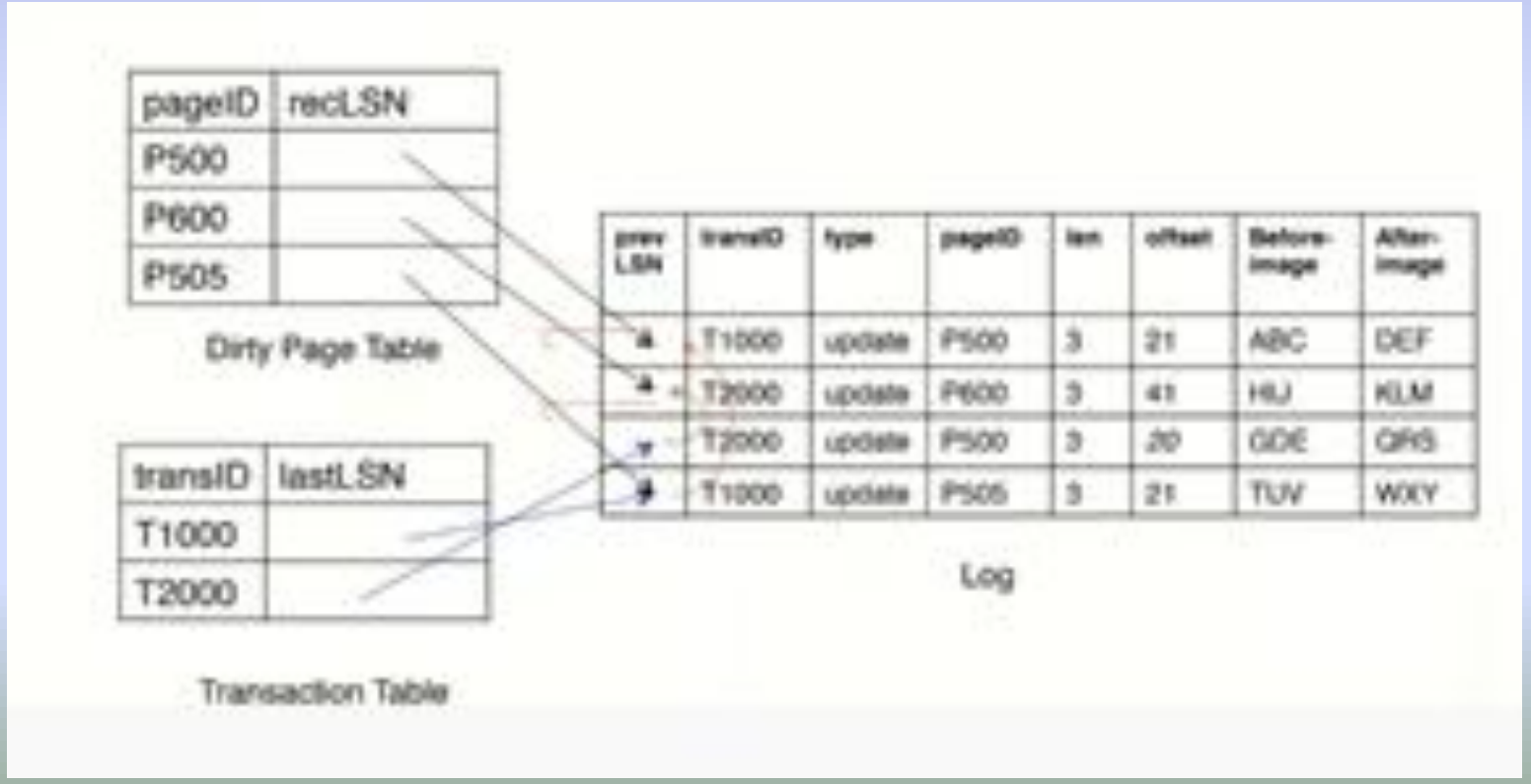
Additional fields for update log records

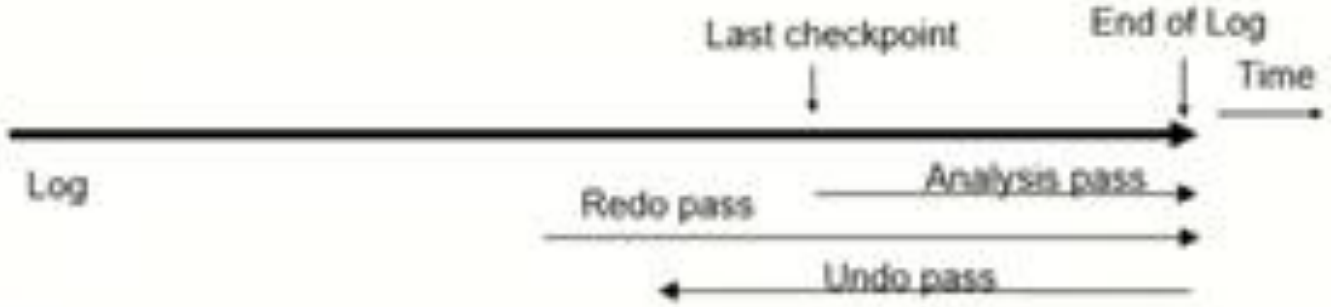
Page ID	Rec LSN
P 500	1012
P 700	1228
P 550	1748

Dirty Page Table

Trans. ID	Last LSN
T 100	1228
T 200	1015
T 350	1786

Transaction Table





- Analysis, redo and undo passes
- Analysis determines where redo should start
- Undo has to go back till start of earliest incomplete transaction

ARIES RECOVERY ALGORITHM

The Analysis phase performs three tasks:

1. It determines the point in the log at which to start the Redo pass.
2. It determines pages in the buffer pool that were dirty at the time of the crash.
3. It identifies Transactions that were active at the time of the crash and must be undone.

Analysis pass

- Starts from last complete checkpoint log record
  - Reads Dirty Page Table from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
  - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record

REDO PHASE

For each redoable log record (update or CLR), the redo phase redoes the change if necessary.

It first checks if the page is in the Dirty Page Table. If it is in the table, then it checks that the recLSN for the page is lower or equal to the LSN of the change under consideration.

If that is the case, then finally the system reads the page from disk and checks if the pageLSN (largest LSN of log written on the page) is strictly smaller than the current LSN.

If that is the case then it redoes the change. Otherwise, it skips the change.

Undo pass: Performs backward scan on log undoing all transaction in undo-list

Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.

At each step pick largest of these LSNs to undo, skip back to it and undo it

If it is of update type write a CLR

After undoing a log record

- For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
- For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record

All intervening records are skipped since they would have been undone already

LSN	LOG
00-08 -	BEGIN CHECK_POINT, END CHECK_POINT
10 -	UPDATE: T1 WRITES P5
20-	UPDATE: T2 WRITES P3
30-	T1 ABORT
40-	CLR: UNDO T1 LSN 10
45-	T1 END
50-	UPDATE: T3 WRITES P1
60-	UPDATE: T2 WRITES P5
x-	CRASH RESTART
70-	CLR: UNDO T2 LSN 60
80-	CLR: UNDO T3 LSN 50
85-	T3 END
x-	CRASH RESTART
90-	CLR: UNDO T2 LSN 20
95-	T2 END

Outlook

- **ARIES data structures**
- ARIES run time behavior
- ARIES recovery algorithm

ARIES Data Structures

Dirty Page Table

Flushed LSN

Transaction Table

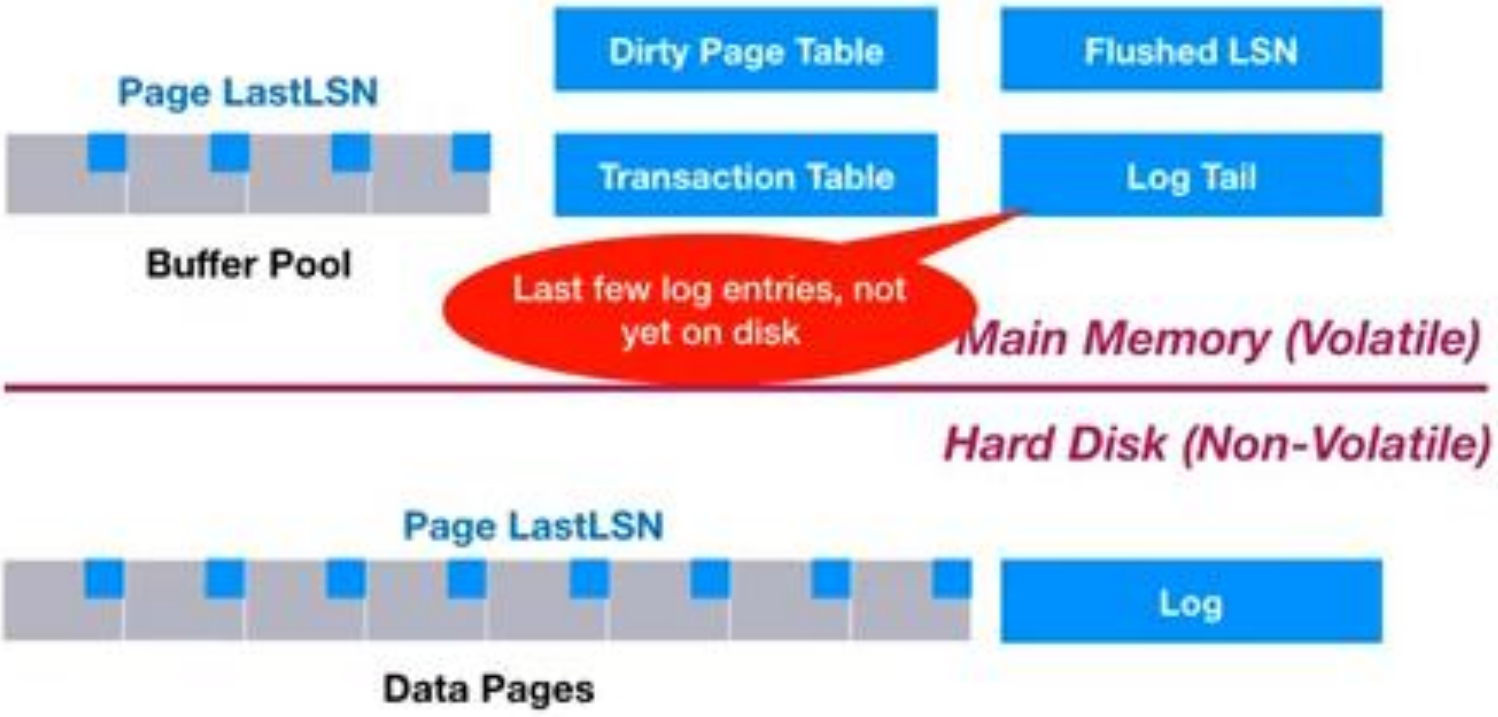
Log Tail

Main Memory (Volatile)

Hard Disk (Non-Volatile)

Log

ARIES Data Structure



ARIES Data Structure

LSN of most recent log entry on hard disk

Dirty Page Table

Flushed LSN

Page LastLSN

Transaction Table

Log Tail

Buffer Pool

Main Memory (Volatile)

Hard Disk (Non-Volatile)

Page LastLSN

Log

Data Pages