How does log based recovery method work? Explain the conditions of using redo and undo operations during the recovery process. [4+4]

10. How does the check point recovery take place in case of transaction failure? Briefly discuss on the shadow paging technique for recovery. [4+4]

11. Write short notes on [4+4]

a) Explain the architecture of remote backup system? [3]

b) What is Immediate database modification technique in context to log based recovery approach? Explain. [4]

Write short notes on:

two phase locking protocol (2PL). [3]

9. Define term Recovery and Atomicity in database. Consider the following log contents when a crash occurs. Explain how recovery would be done for each state. [2+4]

(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>

8. a) Explain the architecture of a remote backup system. [3]

b) What is deferred-database modification technique in context to log based recovery approach? Explain. [3]

8. Write the different types of failures that may occur in system. Differentiate between shadow paging and log-based recovery. [3+3]

Explain the idea of log-based recovery. **[6]**

differs from the view serializability for concurrent execution of transactions. [4+4]

9. What is the purpose of implementing check points in data recovery mechanism? What are the recovery actions performed if failure arises at the end of the given transaction states? [2+4]

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

What is stable storage? Explain the log based recovery mechanism. [2+4]

Write the different types of failures that may occurs in system. Differentiate between shadow paging and log-based recovery. [3+3]

Explain redo phase and undo phase of log based failure recovery mechanism. [6]

9. Distinguish between immediate-modification and deferred-modification in the context of log-based database recovery. What is the significance of checkpoints in a log? [4+2]

9. Briefly explain the idea of a stable storage. Explain the architecture of a remote backup system. [3+3]

). Suppose following contents are present in the log when a crash occurs. Explain what happens for a log-based recovery.

<T_0 start>
<T_0, B, 2000, 2050>
<T_1 start>
<checkpoint {T_0, T_1}>
<T_1, C, 700, 600>
<T_1 commit>
<T_2 start>
<T_2, A, 500, 400>
<T_0, B, 2000>
<T_0 abort>

<T_2, A, 500>
<T_2 abort>

9. Consider the following log contents when a crash occurs. Briefly explain how a recovery would be done. [5]

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>

Failure Classification

1. Transaction Failure:
2. System Crash:
3. Disk Failure:

Transaction Failure Reasons

1. Logical Error: mistakes in the business logic or the way the transaction is formulated.
2. Syntax Error

System Crash Causes

1. Power Failure
2. Fail-Stop Assumption: This is a design assumption in some systems where, upon encountering an error, the system halts its operation rather than continuing in an unpredictable or erroneous state.

Disk Failure Causes

1. Bad Sectors
2. Disk Head Crash

Storage Structure Categories

1. Volatile Storage: This type of storage temporarily holds data. Once the power is turned off, all data is lost. Examples: Main memory (RAM) and cache memory.
2. Non-volatile Storage: This storage retains data even when the system is powered down, making it suitable for long-term data retention.

Examples: Hard disks, magnetic tapes, flash memory.

-> Database recovery refers to restoring the database to a consistent state after a failure.

Log Based Recovery

Log based recovery is a method used in databases to ensure that changes made by transactions can be recovered after a system failure. It works by keeping a detailed log of all operations, so that if something goes wrong, the database can use these logs to restore data to a consistent state.

Key Concepts

1. Log Records:

-> Every action that changes the database is recorded as a log entry. These logs are saved to a stable storage (like a disk) to ensure they are not lost even if the system crashes.

2. Stable Storage:

-> This refers to a reliable storage medium where log records are kept safely. The idea is that even if the system fails, these logs are available for recovery.

3. Transaction Lifecycle in Logging:

a. Start Log:

-> When a transaction begins, it writes a <Ti, Start> record.

-> Example: For a transaction T1, the log will record <T1, Start>

b. Write Log:

-> Before the transaction modifies any data, it writes a log record with the details of the change. This includes:

->Transaction ID (Ti), Data Item (X), Old Value (V1), New Value (V2)

-> Example: If transaction T1 changes a student's city from "ktm" to "Pokhara", the log will record <T1, city, 'ktm', 'Pokhara'>.

c. Commit Log:

-> When the transaction finishes all its operations, it writes a <Ti, Commit> record indicating the successful completion of the transaction.

-> Example: The log records <T1, Commit> once the transaction has finished updating the student's city.

Two Approaches to Log-Based Recovery

1. Deferred Database Modification:
2. Immediate Database Modification:

If a failure happens at any point, the recovery process can refer to these logs to determine what was done and either roll back or reapply the changes to maintain data consistency.

Deferred database modification

Deferred database modification is a way to make sure that a transaction's changes are only applied to the database once the transaction is fully complete and has committed. Instead of immediately updating the actual database, the system first writes all the intended changes to a log kept in stable storage. Only after the transaction reaches a commit state are these logged changes actually applied to the database.

How It Works (Step by Step)

-> When a transaction begins, a "start" record is written to the log (for example, <T1, Start>).

-> Every change the transaction wants to make is logged. For instance, if transaction T1 wants to update a data item X to a new value, it writes a log record like <T1, X, NewValue>. Notice that the log only needs to store the new value because the database won't be changed until later.

-> Even though the log records the change, the actual data in the database is not updated at that moment. The write to the database is "deferred."

-> When the transaction successfully reaches a commit point, a commit record (e.g., <T1, Commit>) is added to the log.
-> Later, either immediately after commit or during a recovery process following a crash, the system reads the log and applies (redoes) the changes to the database.

What Happens in Case of a Crash?

-> If a transaction does not reach the commit stage (for example, it crashes before the commit record is logged), then the deferred changes are simply ignored. Since the database was never updated, there is no need to roll back changes.
-> If a transaction reaches commit before a crash, its log entries are used to "redo" the transaction, ensuring that all changes are eventually applied to the database.

- `<T0, Start>` → Transaction **T0** starts.
- `<T0, A, 950>` → Transaction **T0** wants to update **A** to **950**.
- `<T0, B, 2050>` → Transaction **T0** wants to update **B** to **2050**.
- `<T0, Commit>` → Transaction **T0** successfully commits.
- `<T1, Start>` → Transaction **T1** starts.
- `<T1, C, 600>` → Transaction **T1** wants to update **C** to **600**.
- `<T1, Commit>` → Transaction **T1** successfully commits.

Immediate Database Modification:

-> Database immediately update its stored data while still in the middle of a transaction. However, because these changes are made even before the transaction fully completes (they are "uncommitted"), the system must keep a detailed log so that it can fix things if something goes wrong. The log records both the old value (before the change) and the new value (after the change).

-> When a transaction makes a change, that change is immediately applied to the database. This is called an "immediate update."
-> Since the transaction might not finish (it might later be rolled back or "undone"), these changes are considered "uncommitted."
-> The system records a log entry for each change, saving both the old value (to allow undoing the change) and the new value (to allow redoing it if needed).
-> Undo: This operation restores the data items to their original values if the transaction did not commit. When undoing, the system works backwards through the log.
-> Redo: This operation reapplies the changes from a transaction that successfully committed, working forwards through the log.

Imagine we have two transactions: **T0** and **T1**. The log might look like this:

1. `<T0, Start>`
   Transaction T0 begins.
2. `<T0, A, 1000, 950>`
   T0 changes data item A from 1000 to 950.
3. `<T0, B, 2000, 2050>`
   T0 changes data item B from 2000 to 2050.
4. `<T0, Commit>`
   T0 completes successfully.
5. `<T1, Start>`
   Transaction T1 begins.
6. `<T1, C, 700, 600>`
   T1 changes data item C from 700 to 600.

Now, suppose a crash occurs. The recovery process will do the following:

1. Undo T1:

Since T1 started but never committed (there's no <T1, Commit> record), its changes are reversed. The system goes back through T1's log entry and resets data item C back to its original value, 700.

2. Redo TO:

TO has both a start and a commit record, so its changes are final.
The system goes through TO's log entries in order, reapplying the changes to ensure A is set to 950 and B to 2050.

Feature	Deferred Modification	Immediate Modification
Write Timing	Changes are written to the database **only after** a transaction commits.	Changes can be written to the database **before** the transaction commits.
Log Writing	Logs are recorded **only at commit time.**	Logs are recorded **before each change is applied** to the database.
Undo Requirement	No undo is required since changes are only written after commit.	Undo is required for uncommitted changes if a transaction fails.
Redo Requirement	Redo is required since changes are written only after commit.	Both redo and undo may be required depending on the situation.
System Complexity	Simpler since it only needs redo operations.	More complex due to the need for both undo and redo operations.
Performance	Can be slower since all updates are postponed until commit.	Generally faster since some changes are applied immediately, reducing delay.
Crash Recovery	Easier, as only committed transactions need to be redone.	Harder, as both committed and uncommitted transactions must be handled.
Use Case	Suitable for systems where commit frequency is low, like batch processing.	Suitable for high-performance systems where updates need to be reflected immediately.

Checkpointing in Log-Based Recovery

-> Checkpointing is a process in database recovery that helps reduce the time needed to restore data after a system crash. It marks a point in the log file where all previous changes have been safely written to the database.
-> After a crash, instead of scanning the entire log, the system only checks the records after the last checkpoint, making recovery faster and more efficient.

-> The system scans logs backward to find the last <Checkpoint> record.
-> It identifies active transactions at that checkpoint.
-> Only log records after the checkpoint are considered for recovery.
-> Redo (Reapply Changes): Transactions that committed after the checkpoint are redone.

-> Undo (Revert Changes): Uncommitted transactions are undone (only needed for Immediate Modification).

Shadow paging

-> Shadow paging is a way to help a database recover after a crash without needing a lot of extra work to undo or redo changes.

The system keeps two page tables:

-> The current page table is used during the transaction to track changes.
-> The shadow page table is stored on disk and holds the state of the database before the transaction starts.

During the Transaction:

-> At the start, both tables are exactly the same.
-> Instead of modifying the original database pages directly, the system writes updated pages to a new location on disk.
-> The current page table is updated to point to the newly written pages.
-> The shadow page table remains unchanged during the transaction.

Committing the Transaction:

-> If the transaction completes successfully, the current page table replaces the shadow page table.
-> The database now uses the new modified pages.

Rollback/Failure Handling:

-> If a failure occurs before committing, the system discards the current page table and reverts to the unchanged shadow page table.
-> Since the shadow page table still points to the old, consistent database state, no additional rollback operations are needed.

Advantages of Shadow Paging

✓ **Fast Recovery:** No need for log-based undo/redo operations; simply revert to the shadow page table.

✓ **Consistency Guarantee**: The database always has a consistent state due to the existence of the shadow page table.

✓ **Minimal Overhead**: Eliminates the need for complex logging mechanisms.

Disadvantages of Shadow Paging

✗ **High Storage Overhead**: Requires maintaining multiple copies of pages, increasing disk space usage.

✗ **Fragmentation Issues:** Frequent page replacements lead to database fragmentation, requiring periodic reorganization.

✗ **Inefficiency for Large Databases:** Updating a large page table can be slow and impractical for large-scale databases.

Use Cases

-> Embedded databases

-> File systems (e.g., ZFS)

-> Small-scale database management systems that prioritize fast recovery over performance efficiency

Feature	Log-Based Recovery	Shadow Paging
Approach	Uses logs (undo/redo) to track changes and restore consistency	Uses a shadow page table to maintain a consistent database state
Write Performance	Slower, as logs need to be written before modifying the database	Faster, as changes are written directly to new pages without logging
Read Performance	Faster in normal operations but slower during recovery	May suffer from fragmentation, affecting read speed
Crash Recovery	Requires reading logs and applying undo/redo operations	Simply discards the current page table and reverts to the shadow page table
Complexity	More complex due to log management and recovery procedures	Simpler, as no logs are required for undo/redo operations
Suitability for Large Databases	Well-suited, as logs are efficient for tracking small changes	Less suitable due to storage overhead and fragmentation issues
Example Use Cases	Large-scale databases (e.g., MySQL, PostgreSQL)	Embedded databases, file systems (e.g., ZFS), small DBMS

Logical Undo vs. Physical Undo

-> **Physical Undo**: Restores a record to its previous value. For example, if a value changes from 10 to 15, physical undo would simply change 15 back to 10.

-> **Logical Undo**: Instead of reverting values directly, it performs an opposite operation. For instance, if a new record was inserted, the undo would delete that record.

How Recovery Works After a Crash

When a crash happens, the system goes through two main phases:

1. **Redo Phase:** The system re-executes the actions from the log that completed normally. This ensures that all changes intended to be committed are applied again.

2. **Undo Phase**: The system then scans the log in reverse order, undoing any operations that did not complete fully. If a transaction was interrupted:

9. Define term Recovery and Atomicity in database. Consider the following log contents when a crash occurs. Explain how recovery would be done for each state. [3] [2+4]

(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>

a → T0 not commit undo save

b → T0 → commited (redo) T1 → undo

c → T0 → redo , T1 → redo