KARNAVATI UNIVERSITY | UIT

Unitedworld Institute Of Technology

B.Tech. Computer Science & Engineering

Semester : 3rd

Introduction To Database Management System
Course Code: 71203002003

Unit IV : TRANSACTIONS, CONCURRENCY CONTROL AND DEADLOCKS

Prepared by:
Mr. Utsav Kapadiya
Assistant Professor (UIT)

Recovery from Transaction Failures in DBMS

Understanding how database management systems maintain data integrity and consistency when failures occur is crucial for building reliable applications. This presentation explores the fundamental concepts, techniques, and mechanisms that enable DBMS to recover from various types of transaction failures.

What is Transaction Failure?

A transaction failure occurs when a transaction cannot complete its execution successfully. When this happens, the database must be restored to a consistent state to prevent data corruption and maintain integrity. Transaction failures can happen for various reasons, from logical errors in the code to hardware malfunctions.
The database management system must have mechanisms in place to detect these failures and take appropriate action—either undoing incomplete changes or reapplying committed ones—to ensure the database remains in a valid state.

Example: Bank Transfer Transaction

Consider a simple bank transaction where ₹1000 is transferred from Account A to Account B. This seemingly simple operation involves multiple steps that must all complete successfully.

1

Read Account A

Retrieve current balance of Account A

2

Deduct Amount

Calculate: A = A - 1000

3

Write Account A

Update Account A with new balance

4

Read Account B

Retrieve current balance of Account B

5

Add Amount

Calculate: B = B + 1000

6

Write Account B

Update Account B with new balance

7

Commit

Finalize the transaction

Critical Problem: If a failure occurs after deducting ₹1000 from Account A but before adding it to Account B, the money disappears from the system entirely—creating an inconsistent state that violates the fundamental principle of conservation of funds.

Types of Transaction Failures

Database systems can experience various types of failures, each requiring different recovery approaches. Understanding these failure types helps DBAs implement appropriate safeguards and recovery strategies.

Logical Errors

Transaction cannot complete due to logical conditions or business rule violations

- Negative balance detected
- Constraint violations
- Deadlock situations
- Invalid data types

System Crash

System failure occurs before transaction completes successfully

- Power outages
- Operating system crashes
- Software bugs
- Hardware failures

Media Failure

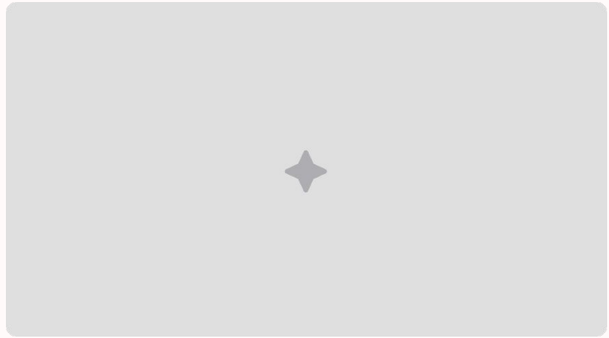
Physical disk failure or data corruption on storage devices

- Hard disk crashes
- Disk controller failure
- Data corruption
- Storage device malfunction

What is Database Recovery?

Recovery is the process of restoring the database from an inconsistent or corrupted state back to a consistent, reliable state after a failure. This critical function ensures that committed transactions are never lost and incomplete transactions don't leave the database in an invalid state.

The DBMS employs sophisticated algorithms and data structures to track changes, maintain logs, and coordinate recovery operations. These mechanisms work together to provide the ACID properties (Atomicity, Consistency, Isolation, Durability) that are fundamental to reliable database operations.



Log-Based Recovery

Uses transaction logs to track all database modifications and replay or undo operations as needed

Checkpoint Mechanisms

Creates periodic snapshots to minimize recovery time by limiting log scanning

Write-Ahead Logging Protocol

The Write-Ahead Logging (WAL) protocol is the golden rule of log-based recovery and forms the foundation of modern database recovery systems. This protocol ensures that the database can always be recovered to a consistent state, even in the event of sudden failures.

> **The WAL Rule:** Before making any change to the database, the corresponding log record must be written to the log file. This ensures that even if a crash happens immediately after a change, the DBMS can recover it using the log.

By maintaining this discipline, the DBMS creates a reliable audit trail of all modifications. If the system crashes, these log records serve as the blueprint for reconstructing the database to its correct state—either by reapplying committed changes or undoing incomplete ones.

Anatomy of a Log File

A log file is a sequential record stored on stable storage that captures every significant action performed by transactions. This file is separate from the database itself, providing an independent record that survives crashes affecting the main database.

Transaction Start

```
<Tn, start>
```

Marks the beginning of transaction Tn

Data Modification

```
<Tn, X, old_value, new_value>
```

Records changes to data item X with before and after values

Transaction Commit

```
<Tn, commit>
```

Indicates successful completion of transaction Tn

Transaction Abort

```
<Tn, abort>
```

Marks rollback of transaction Tn due to error

Log Entry Example

Let's examine how log entries capture the bank transfer transaction from our earlier example. Each operation generates a corresponding log record before the actual database modification occurs.

Log Record	Meaning
`<T1, start>`	Transaction T1 has begun execution
`<T1, A, 1000, 900>`	Account A balance changed from ₹1000 to ₹900
`<T1, B, 2000, 2100>`	Account B balance changed from ₹2000 to ₹2100
`<T1, commit>`	Transaction T1 completed successfully

This log provides a complete history of the transaction. If a failure occurs, the recovery manager examines these entries to determine the appropriate action: if the commit record exists, the changes should be redone; if not, they should be undone.

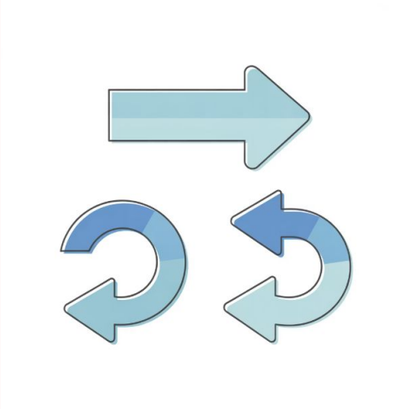
UNDO and REDO Operations

UNDO Operation

Used for uncommitted transactions that were active when failure occurred. The UNDO operation reverses the effects of incomplete transactions by restoring old values from the log.

Example: If T2 was still running when the system crashed, we must remove all its changes to return the database to the state before T2 started.

REDO Operation



Used for committed transactions whose changes may not have been written to disk. The REDO operation reapplies operations using new values from the log.

Example: If T1 committed but the crash occurred before writing to disk, we must reapply all T1's changes to ensure they persist.

Recovery Decision Rule: If a transaction has a commit record in the log before the failure, it must be REDONE. If a transaction started but has no commit record, it must be UNDONE.

Understanding Checkpoints

A checkpoint is a synchronization point where the DBMS ensures that all modified data in memory (buffer) is written to disk, and the system's current state is recorded in the log. Think of it as taking a snapshot of the database at a specific moment in time.

Checkpoints dramatically improve recovery efficiency. Without checkpoints, the recovery process would need to scan the entire log file from the beginning—potentially millions of entries. With checkpoints, recovery only needs to examine entries after the most recent checkpoint, reducing recovery time from hours to minutes or even seconds.

Flush Dirty Pages

All modified data pages in memory are written to disk storage

Write Log Records

All log records in memory buffers are flushed to the log file

Record Checkpoint

A checkpoint entry with active transactions is added to the log

The Checkpoint Advantage

Without checkpoints, recovery becomes a time-consuming nightmare. With checkpoints, it becomes a manageable process.

10K+

Without Checkpoint

Log entries to scan from database start to failure point

500

With Checkpoint

Log entries to scan from last checkpoint to failure point

95%

Time Saved

Reduction in recovery time using checkpoint mechanism

Checkpoint Log Entry Format

When a checkpoint occurs, the DBMS writes a special entry to the log that records which transactions were active at that moment. This information is crucial for the recovery process.

```
<Checkpoint, {T1, T2, T3}>
```

This entry indicates that transactions T1, T2, and T3 were in progress when the checkpoint was taken. During recovery, the system knows it must examine what happened to these transactions after the checkpoint.

Before Checkpoint

Multiple transactions executing, logs accumulating in memory

1

2

Checkpoint Occurs

<Checkpoint, {T1, T2}> written, all buffers flushed to disk

After Checkpoint

New transactions begin, system continues normal operation

3

4

System Crash

Recovery starts from checkpoint, not from beginning

Types of Checkpoints

Sharp Checkpoint

A sharp checkpoint stops all transaction processing while it writes everything to disk. This creates a precise, consistent snapshot but causes a brief pause in database operations.

Advantages: Simple to implement, creates perfect consistency point

Disadvantages: System becomes temporarily unavailable, not suitable for high-traffic databases

Fuzzy Checkpoint

A fuzzy checkpoint allows transactions to continue while checkpointing happens gradually in the background. Modern DBMS like Oracle and MySQL use this approach.

Advantages: No system downtime, minimal performance impact

Disadvantages: More complex to implement, requires careful coordination

Step-by-Step Recovery Process

When a system failure occurs, the DBMS initiates a systematic recovery procedure that uses the log file and checkpoint information to restore database consistency.

Locate Last Checkpoint

Scan backward through the log file to find the most recent checkpoint entry

Identify Transaction States

Determine which transactions committed and which were incomplete at failure

Build Transaction Lists

Create REDO list (committed) and UNDO list (uncommitted) for processing

Execute UNDO Operations

Roll back all incomplete transactions using old values from the log

Execute REDO Operations

Reapply all committed transactions using new values from the log

Log Recovery Actions

Write log entries for all undo/redo operations to maintain audit trail

Recovery Example: Transaction States

Consider a scenario where the system crashes while multiple transactions are in various stages of execution. The log file contains the following entries:

```
<T1, start><T1, A, 100, 90><T2, start><T2, B, 50, 60><T1, commit><Checkpoint, {T2}><T3, start><T2, B, 60, 70><T3, C, 200, 250><System Crash>
```

Transaction T1

Status: Committed before checkpoint

Action: REDO to ensure changes persist

Transaction T2

Status: Active at checkpoint, no commit

Action: UNDO all modifications

Transaction T3

Status: Started after checkpoint, no commit

Action: UNDO all modifications

Complete Recovery Techniques

Database management systems employ multiple recovery strategies, each suited to different failure scenarios and operational requirements. Understanding when to use each technique is essential for database administrators.

Log-Based Recovery

Uses transaction logs to undo or redo changes after failures. The foundation of modern DBMS recovery.

- Immediate recovery capability
- Transaction-level granularity
- Supports ACID properties

Checkpoint Mechanism

Creates periodic snapshots to minimize log scanning during recovery.

- Reduces recovery time significantly
- Optimizes log file management
- Balances performance and reliability

Shadow Paging

Maintains shadow copies of database pages before modifications are made.

- No undo logging required
- Atomic page updates
- Higher storage overhead

Backup and Restore

Periodically creates complete database copies for recovery from catastrophic failures.

- Protection against media failures
- Long-term data preservation
- Disaster recovery foundation

Checkpoint Frequency Strategy

Determining the optimal checkpoint frequency requires balancing recovery time against system performance. Too frequent checkpoints cause overhead; too infrequent checkpoints slow recovery.

Time-Based Checkpoints

Execute checkpoints at regular intervals (e.g., every 5 or 10 minutes) regardless of activity level

Transaction-Based Checkpoints

Trigger checkpoint after processing a specific number of transactions (e.g., every 1000 transactions)

Log-Size-Based Checkpoints

Initiate checkpoint when log file reaches a predetermined size threshold (e.g., 100 MB)

Adaptive Checkpoints

Dynamically adjust checkpoint frequency based on system load and transaction patterns

Advantages and Limitations

Key Advantages

- **Ensures Atomicity**

  Either all operations of a transaction occur or none—no partial results

- **Maintains Consistency**

  Database always returns to a valid state after any failure

- **Fast Recovery**

  Checkpoints dramatically reduce time needed to restore database

- **Automatic Operation**

  Recovery happens automatically without manual intervention

- **Audit Trail**

  Complete history of all database modifications for compliance

Important Limitations

- **Storage Overhead**

  Log files can grow very large, consuming significant disk space

- **Performance Impact**

  Writing logs before every change adds latency to transactions

- **Complexity**

  Sophisticated algorithms required for distributed database recovery

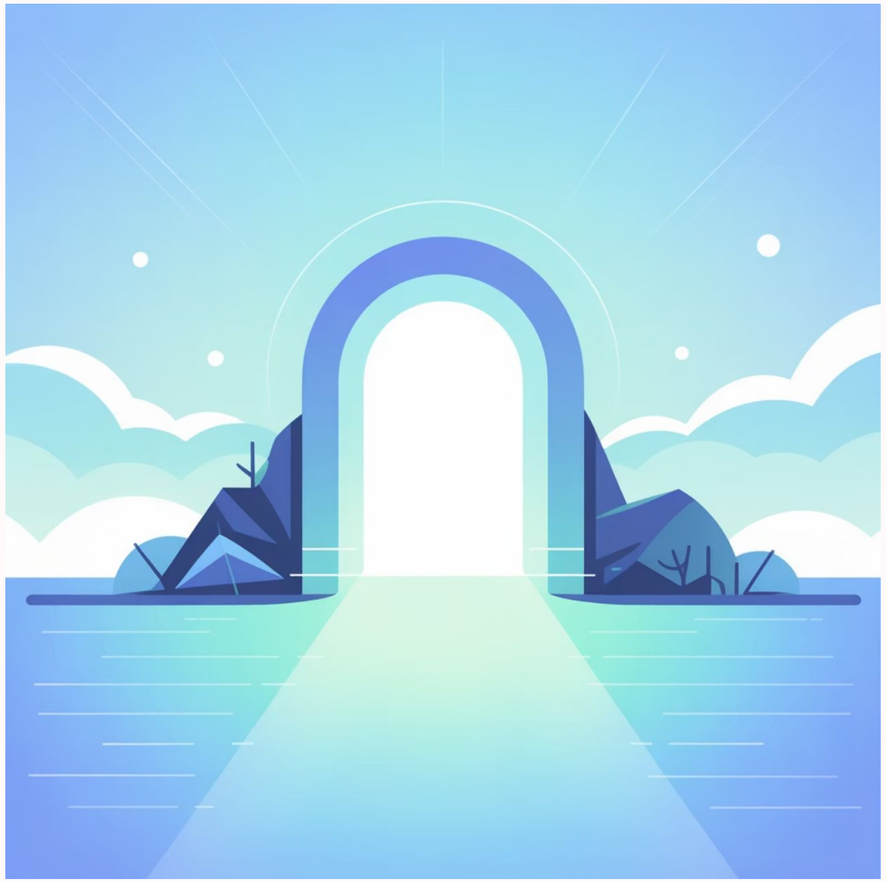
- **Log Management**

  Requires strategies for log archiving, backup, and pruning

- **Recovery Time**

  Large transactions can still require substantial time to undo/redo

Real-World Analogy



The Video Game Save Point

Think of database recovery mechanisms like save points in a video game. When playing, you regularly save your progress at checkpoints. If the game crashes or you make a mistake, you don't lose everything—you restart from your last save point.
Similarly, in DBMS:

The game state = Database contents

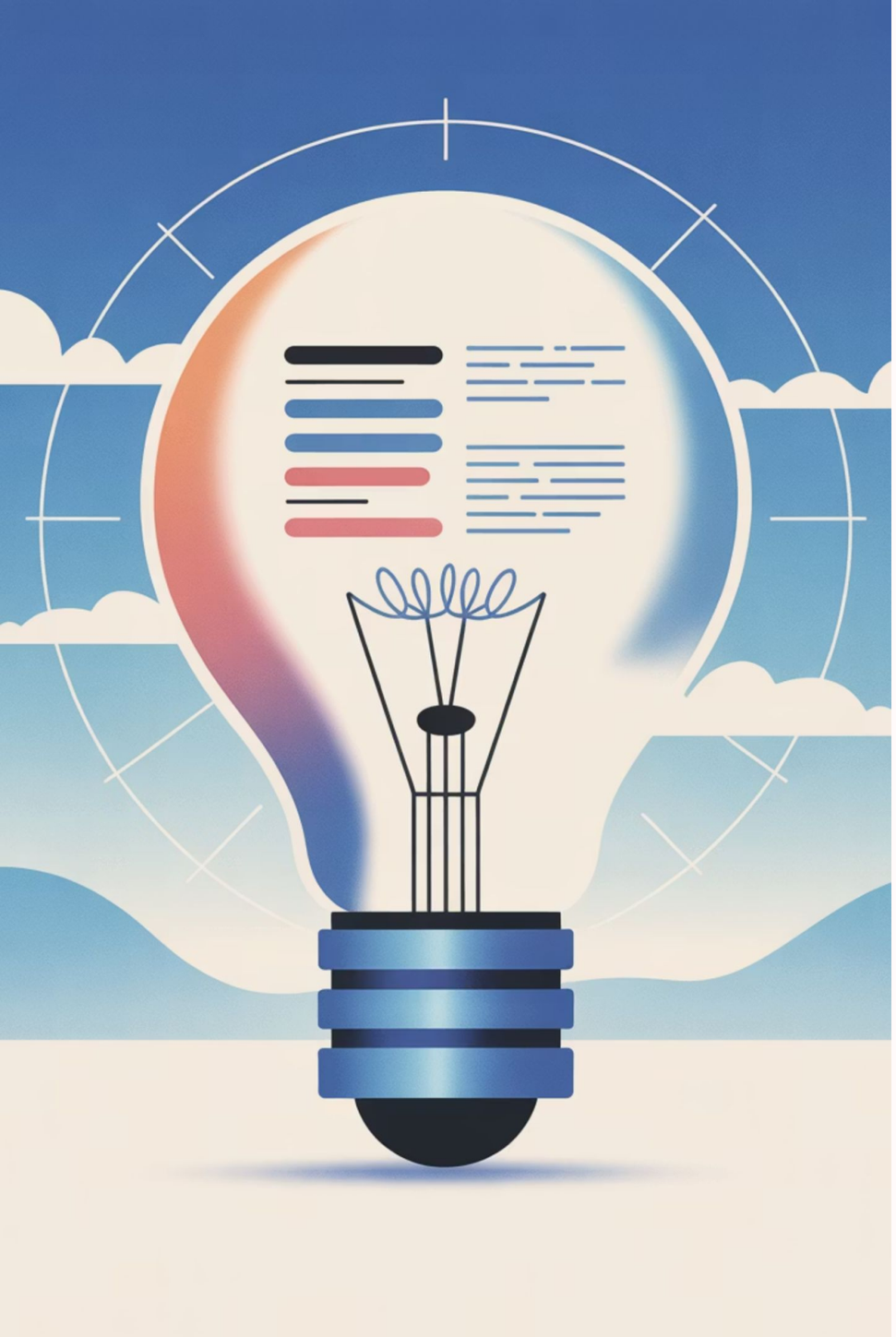
Save points = Checkpoints

Save file = Log file

Game crash = System failure

Reload = Recovery process

Just as gamers appreciate save points that let them resume quickly after failures, database users rely on recovery mechanisms to protect their data and minimize downtime.



Key Takeaways

Reliability First

Recovery mechanisms are not optional—they're fundamental to database integrity and essential for production systems.

Write-Ahead Logging

The WAL protocol ensures that every change is logged before being applied, creating a reliable foundation for recovery.

Smart Checkpointing

Regular checkpoints balance system performance with recovery speed, making restoration practical even for large databases.

UNDO and REDO

These complementary operations work together to handle both committed and uncommitted transactions during recovery.

Understanding transaction recovery is crucial for anyone working with databases. These mechanisms operate silently in the background, but they're what make modern database systems reliable enough to store critical data for banking, healthcare, e-commerce, and countless other applications.

Thank You!