# Chapter

# Database Recovery Techniques

**Dr. Ashim Saha,**
**CSED, NIT Agartala**

# Chapter Outline

**Databases Recovery**

1. Purpose of Database Recovery
2. Types of Failure
3. Transaction Log
4. Data Updates
5. Data Caching
6. Transaction Roll-back (Undo) and Roll-Forward
7. Checkpointing
8. Recovery schemes
9. ARIES Recovery Scheme
10. Recovery in Multidatabase System

# Database Recovery

## 1 Purpose of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).

Example: If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

# Database Recovery

## 2 Types of Failure

The database may become unavailable for use due to

- Transaction failure: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
- System failure: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- Media failure: Disk head crash, power disruption, etc.

# Database Recovery

## 3 Transaction Log

For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required. These values and other information is stored in a sequential file called Transaction log. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

| T ID | Back P | Next P | Operation | Data item | BFIM | AFIM |
|---|---|---|---|---|---|---|
| T1 | 0 | 1 | Begin | | | |
| T1 | 1 | 4 | Write | X | X = 100 | X = 200 |
| T2 | 0 | 8 | Begin | | | |
| T1 | 2 | 5 | W | Y | Y = 50 | Y = 100 |
| T1 | 4 | 7 | R | M | M = 200 | M = 200 |
| T3 | 0 | 9 | R | N | N = 400 | N = 400 |
| T1 | 5 | nil | End | | | |

- **[start_transaction, T3]**
- **[read_item, T3, C]**
- **[write_item, T3, B, 15, 12]**
- **[start_transaction,T2]**
- **[read_item, T2, B]**
- **[write_item, T2, B, 12, 18]**
- **[start_transaction,T1]**
- **[read_item, T3, A]**
- **[read_item, T3, D]**
- **[write_item, T1, D, 20, 25]**
-

- **[start_transaction, T3]**
- **[read_item, T3, C]**
- **[write_item, T3, B, 15, 12]**
- **[start_transaction,T2]**
- **[read_item, T2, B]**
- **[write_item, T2, B, 12, 18]**
- **[start_transaction,T1]**
- **[read_item, T3, A]**
- **[read_item, T3, D]**
- **[write_item, T1, D, 20, 25]**
-

| Sl | T_ID | BackP | NextP | Operation | Data Item | BFIM | AFIM |
|---|---|---|---|---|---|---|---|
| 1 | T3 | 0 | 2 | B | | | |
| 2 | T3 | 1 | 3 | R | C | 50 | 50 |
| 3 | T3 | 2 | 8 | W | B | 15 | 12 |
| 4 | T2 | 0 | 5 | B | | | |
| 5 | T2 | 4 | 6 | R | B | 12 | 12 |
| 6 | T2 | 5 | 11 | W | B | 12 | 18 |
| 7 | T1 | 0 | 10 | B | | | |
| 8 | T3 | 3 | 9 | R | A | 40 | 40 |
| 9 | T3 | 8 | 13 | R | D | 20 | 20 |
| 10 | T1 | 7 | 12 | W | D | 20 | 25 |

# Database Recovery

**4 Data Update**

- **Immediate Update**: As soon as a data item is modified in cache, the disk copy is updated.
- **Deferred Update**: All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- **Shadow update**: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- **In-place update**: The disk version of the data item is overwritten by the cache version.

# Database Recovery

## 5 Data Caching

Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk. The flushing is controlled by **Modified** and **Pin-Unpin** bits.

**Pin-Unpin**: Instructs the operating system not to flush the data item.

**Modified**: Indicates the AFIM of the data item.

# Database Recovery

**6 Transaction Roll-back (Undo) and Roll-Forward (Redo)**

To maintain atomicity, a transaction's operations are **redone** or **undone**.

**Undo**: Restore all BFIMs on to disk (Remove all AFIMs).

**Redo**: Restore all AFIMs on to disk.

Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.

# Database Recovery

**Roll-back**

We show the process of roll-back with the help of the following three transactions T1, and T2 and T3.

| T1 | T2 | T3 |
|---|---|---|
| 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 (A) | write_item (A) |

# Database Recovery

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

| | | A | B | C | D |
|---|---|---|---|---|---|
| | | 30 | 15 | 40 | 20 |
| | [start_transaction, T3] | | | | |
| | [read_item, T3, C] | | | | |
| * | [write_item, T3, B, 15, 12] | | 12 | | |
| | [start_transaction,T2] | | | | |
| | [read_item, T2, B] | | | | |
| ** | [write_item, T2, B, 12, 18] | | 18 | | |
| | [start_transaction,T1] | | | | |
| | [read_item, T1, A] | | | | |
| | [read_item, T1, D] | | | | |
| | [write_item, T1, D, 20, 25] | | | | 25 |
| | [read_item, T2, D] | | | | |
| | [write_item, T2, D, 25, 26] | | | | 26 |
| | [read_item, T3, A] | | | | |

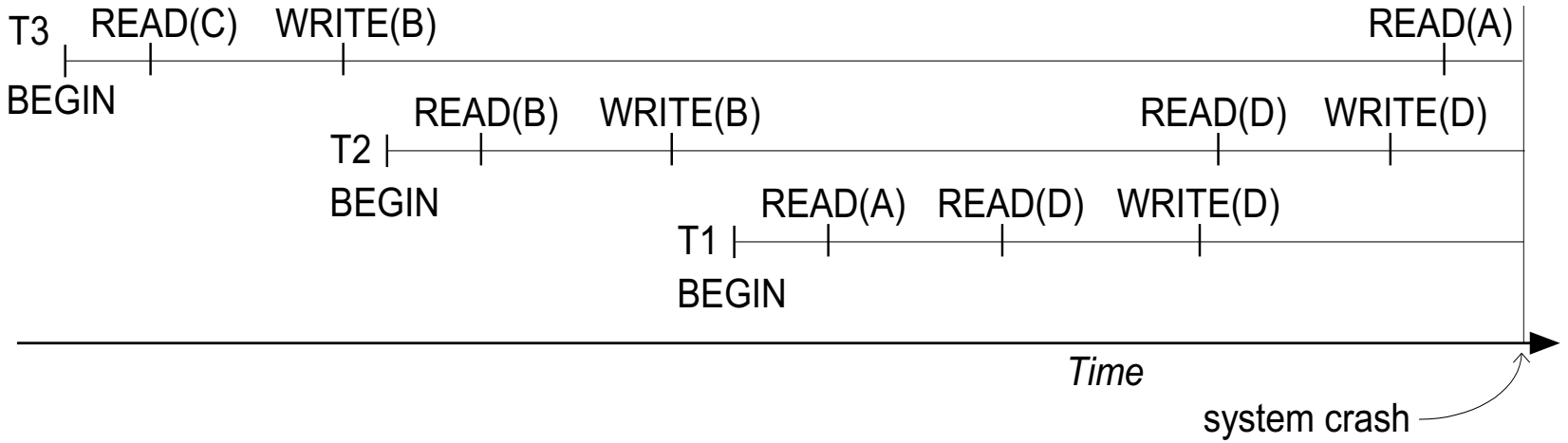
---- T3 failed system crash ----

* T3 is rolled back because it did not reach its commit point.

** T2 is rolled back because it reads the value of item B written by T3.

# Database Recovery

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




**Illustrating cascading roll-back**

# Database Recovery

**Write-Ahead Logging**

When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging** (WAL) protocol. WAL states that

**For Undo**: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).

**For Redo**: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

# Database Recovery

## 7 Checkpointing

Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps defines a checkpoint operation:

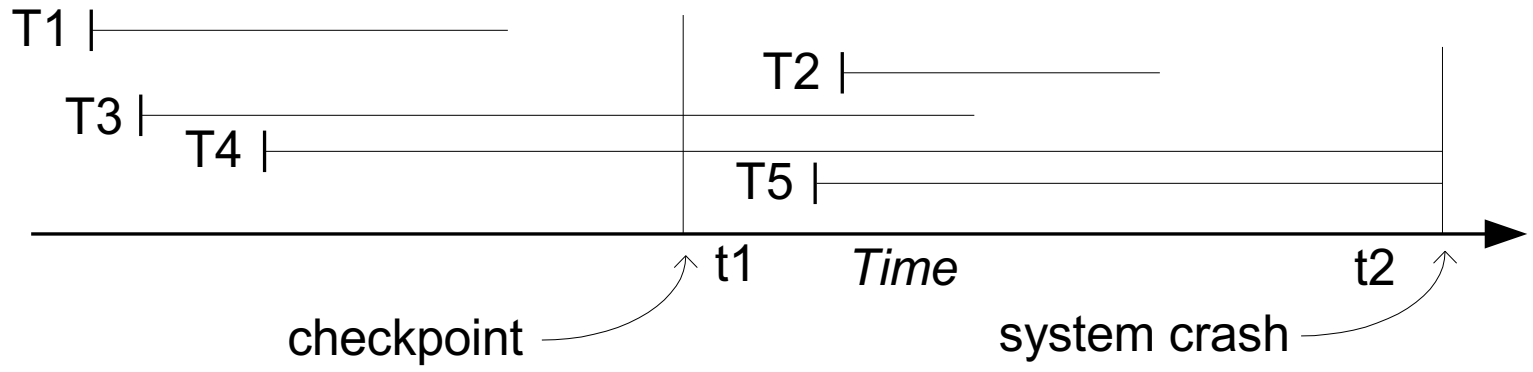
1. Suspend execution of transactions temporarily.
2. Force write modified buffer data to disk.
3. Write a [checkpoint] record to the log, save the log to disk.
4. Resume normal transaction execution.

During recovery **redo** or **undo** is required to transactions appearing after [checkpoint] record.

# Database Recovery

## Deferred Update with concurrent users

This environment requires some concurrency control mechanism to guarantee **isolation** property of transactions. In a system recovery transactions which were recorded in the log after the last checkpoint were **redone**. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.




**Recovery in a concurrent users environment.**

# Database Recovery

**Steal/No-Steal and Force/No-Force**

Possible ways for flushing database cache to database disk:

**Steal**: Cache can be flushed before transaction commits.

**No-Steal**: Cache cannot be flushed before transaction commit.

**Force**: Cache is immediately flushed (forced) to disk.

**No-Force**: Cache is deferred until transaction commits.

These give rise to four different ways for handling recovery:

Steal/No-Force (Undo/Redo), Steal/Force (Undo/No-redo), No-Steal/No-Force (No-Undo/Redo) and No-Steal/Force (No-undo/No-redo).

# Database Recovery

**Recovery Scheme**

**Deferred Update (No Undo/Redo)**

The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

# Database Recovery

**Deferred Update in a single-user system**

There is no concurrent data sharing in a single user system. The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

# Database Recovery

**Deferred Update in a concurrent system**

(a)

| T1 | T2 |
|---|---|
| read_item (A) | read_item (B) |
| read_item (D) | write_item (B) |
| write_item (D) | read_item (D) |
| | write_item (A) |

(b)

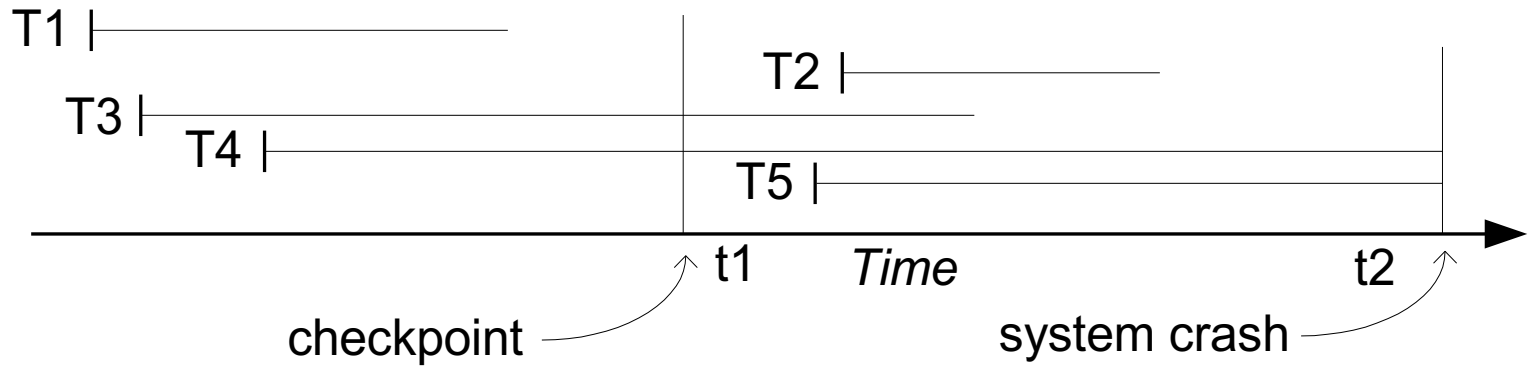
[start_transaction, T1]
[write_item, T1, D, 0,20]
[commit T1]
[start_transaction, T2]
[write_item, T2, B, 0,10]
[write_item, T2, D,20, 25] ← system crash

The operations of T1 will be redo, T2 are to be also redo.

# Database Recovery

## Deferred Update with concurrent users

This environment requires some concurrency control mechanism to guarantee **isolation** property of transactions. In a system recovery transactions which were recorded in the log after the last checkpoint were **redone**. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.




**Recovery in a concurrent users environment.**

# Database Recovery

## Deferred Update with concurrent users

(a)

| T1 | T2 | T3 | T4 |
|---|---|---|---|
| read_item (A) | read_item (B) | read_item (A) | read_item (B) |
| read_item (D) | write_item (B) | write_item (A) | write_item (B) |
| write_item (D) | read_item (D) | read_item (C) | read_item (A) |
| | write_item (D) | write_item (C) | write_item (A) |

(b)

[start_transaction, T1]
[write_item, T1, D, 20]
[commit, T1]
[checkpoint]
[start_transaction, T4]
[write_item, T4, B, 15]
[write_item, T4, A, 20]
[commit, T4]
[start_transaction T2]
[write_item, T2, B, 12]
[start_transaction, T3]
[write_item, T3, A, 30]
[write_item, T2, D, 25] ← system crash

T2 and T3 are ignored because they did not reach their commit points.
T4 is redo because its commit point is after the last checkpoint.

# Database Recovery

## Deferred Update with concurrent users

Two tables are required for implementing this protocol:

**Active table**: All active transactions are entered in this table.
**Commit table**: Transactions to be committed are entered in this table.

During recovery, all transactions of the **commit** table are redo and all transactions of **active** tables are ignored since none of their AFIMs reached the database. It is possible that a **commit** table transaction may be **redo** twice but this does not create any inconsistency because of a redone is "**idempotent**", that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

# Database Recovery

## Recovery Techniques Based on Immediate Update

### Undo/No-redo Algorithm

In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits. For this reason the recovery manager **undoes** all transactions during recovery. No transaction is **redo**. It is possible that a transaction might have completed execution and ready to commit but this transaction is also **undo**.

# Database Recovery

## Recovery Techniques Based on Immediate Update

### Undo/Redo Algorithm (Single-user environment)

Recovery schemes of this category apply **undo** and also **redo** for recovery. In a single-user environment no concurrency control is required but a log is maintained under WAL. Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table. The recovery manager performs:

1. **Undo** of a transaction if it is in the **active** table.
2. **Redo** of a transaction if it is in the **commit** table.

# Database Recovery

## Recovery Techniques Based on Immediate Update

### Undo/Redo Algorithm (Concurrent execution)

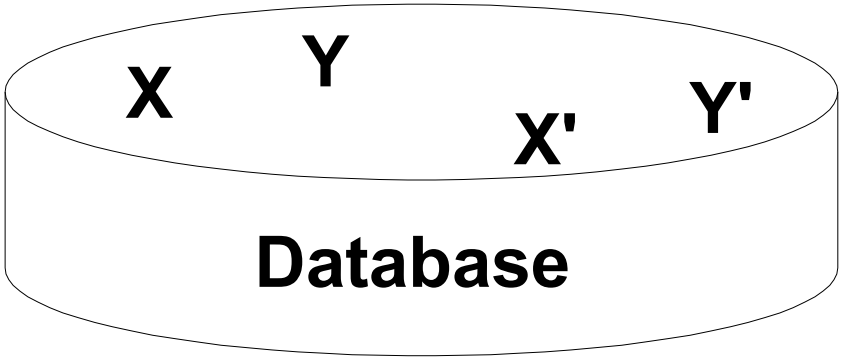
Recovery schemes of this category applies undo and also redo to recover the database from failure. In concurrent execution environment a concurrency control is required and log is maintained under WAL. Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager checkpointing is used. The recovery performs:

1. **Undo** of a transaction if it is in the active table.
2. **Redo** of a transaction if it is in the commit table.

# 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.



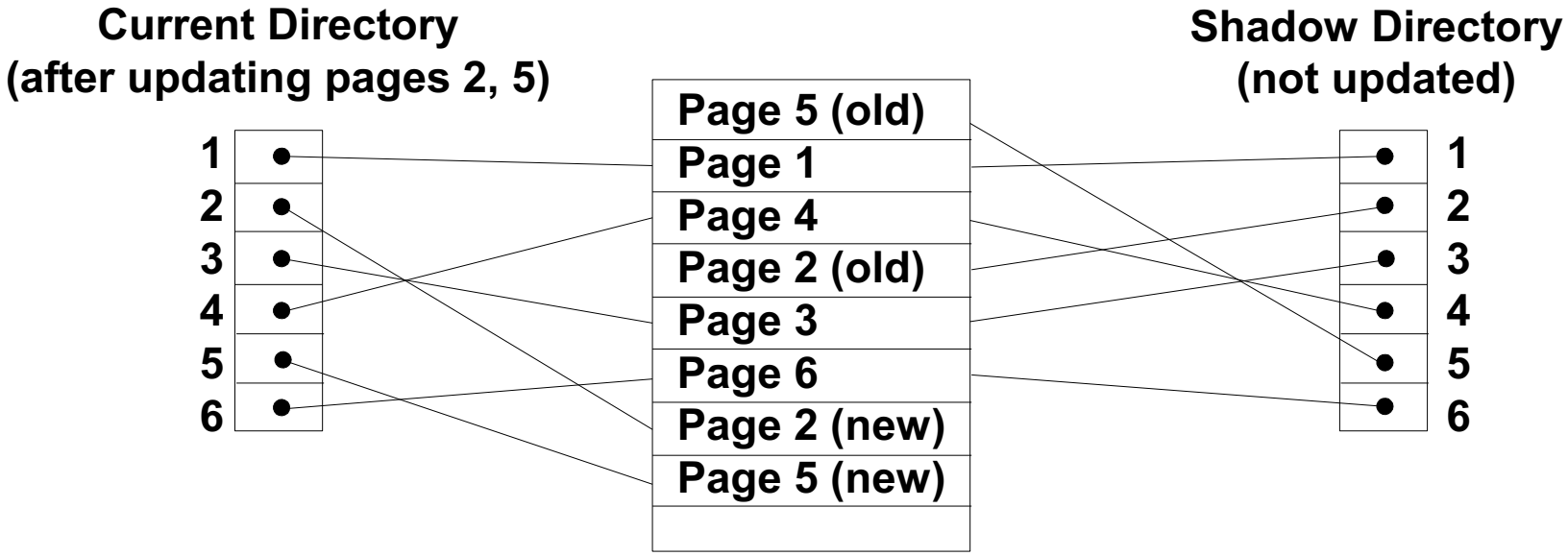

X and Y: Shadow copies of data items

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.



# Database Recovery

## 9 The ARIES Recovery Algorithm

The ARIES Recovery Algorithm is based on:

1. WAL (Write Ahead Logging)
2. Repeating history during redo: ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred.
3. Logging changes during undo: It will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

# Database Recovery

## The ARIES Recovery Algorithm

The ARIES recovery algorithm consists of three steps:

1. **Analysis**: step identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of crash. The appropriate point in the log where redo is to start is also determined.
2. **Redo**: necessary redo operations are applied.
3. **Undo**: log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.

# Database Recovery

## The ARIES Recovery Algorithm

### The Log and Log Sequence Number (LSN)

A log record is written for (a) data update, (b) transaction commit, (c) transaction abort, (d) undo, and (e) transaction end. In the case of undo a compensating log record is written.

A unique LSN is associated with every log record. LSN increases monotonically and indicates the disk address of the log record it is associated with. In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.

A log record stores (a) the previous LSN of that transaction, (b) the transaction ID, and (c) the type of log record.

# Database Recovery

## The ARIES Recovery Algorithm

### The Log and Log Sequence Number (LSN)

A log record stores:

1. Previous LSN of that transaction: It links the log record of each transaction. It is like a back pointer points to the previous record of the same transaction.
2. Transaction ID
3. Type of log record.

For a write operation the following additional information is logged:

4. Page ID for the page that includes the item
5. Length of the updated item
6. Its offset from the beginning of the page
7. BFIM of the item
8. AFIM of the item

# Database Recovery

## The ARIES Recovery Algorithm

### The Transaction table and the Dirty Page table

For efficient recovery following tables are also stored in the log during checkpointing:

**Transaction table**: Contains an entry for each active transaction, with information such as transaction ID, transaction status and the LSN of the most recent log record for the transaction.

**Dirty Page table**: Contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

# Database Recovery

## The ARIES Recovery Algorithm

### Checkpointing

A checkpointing does the following:

1. Writes a *begin_checkpoint* record in the log
2. Writes an *end_checkpoint* record in the log. With this record the contents of transaction table and dirty page table are appended to the end of the log.
3. Writes the LSN of the *begin_checkpoint* record to a special file. This special file is accessed during recovery to locate the last checkpoint information.

To reduce the cost of checkpointing and allow the system to continue to execute transactions, ARIES uses "fuzzy checkpointing".

# Database Recovery

## The ARIES Recovery Algorithm

The following steps are performed for recovery

1. **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.
2. **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.
3. **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.

# Database Recovery

## An example of the working of ARIES scheme

(a)

| LSN | LAST-LSN | TRAN-ID | TYPE | PAGE-ID | Other Info. |
|---|---|---|---|---|---|
| 1 | 0 | T1 | update | C | ----- |
| 2 | 0 | T2 | update | B | ----- |
| 3 | 1 | T1 | commit | | ----- |
| 4 | begin checkpoint | | | | |
| 5 | end checkpoint | | | | |
| 6 | 0 | T3 | update | A | ----- |
| 7 | 2 | T2 | update | C | ----- |
| 8 | 7 | T2 | commit | | ----- |

(b)

**TRANSACTION TABLE**

| TRANSACTION ID | LAST LSN | STATUS |
|---|---|---|
| T1 | 3 | commit |
| T2 | 2 | in progress |

**DIRTY PAGE TABLE**

| PAGE ID | LSN |
|---|---|
| C | 1 |
| B | 2 |

(c)

**TRANSACTION TABLE**

| TRANSACTION ID | LAST LSN | STATUS |
|---|---|---|
| T1 | 3 | commit |
| T2 | 8 | commit |
| T3 | 6 | in progress |

**DIRTY PAGE TABLE**

| PAGE ID | LSN |
|---|---|
| C | 1 |
| B | 2 |
| A | 6 |

# Database Recovery

## 10 Recovery in multidatabase system

A multidatabase system is a special distributed database system where one node may be running relational database system under Unix, another may be running object-oriented system under window and so on. A transaction may run in a distributed fashion at multiple nodes. In this execution scenario the transaction commits only when all these multiple nodes agree to commit individually the part of the transaction they were executing. This commit scheme is referred to as "*two-phase commit*" (2PC). If any one of these nodes fails or cannot commit the part of the transaction, then the transaction is aborted. Each node recovers the transaction under its own recovery protocol.