# Recovery System

DBMS

# Outline

- Recovery?
- Importance of Recovery
- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Remote Backup System
- Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)
- RAID (Left on the students)

## What is Recovery?

- It is the process of restoring the db to most recent consistent state that existed just before the failure.
- It is the responsibility of recovery manager/ recovery management component.
- 3 states of database recovery
  - Pre-condition {consistent state}
  - Condition {failure occur}
  - Post-condition {restoring the db to most recent consistent state that existed just before the failure}

## Why Recovery is important?

- A DBMS has to ensure ACID properties even in the case, a transaction fail due to system crash or failure
  - Mainly atomicity and durability.
  - Otherwise DB may lead to inconsistent state
- How to recover?
  - Backup and restore
  - Recovery algorithms
- An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure.

## Failure Reason

- System failure occur due to variety of reasons such as disk crash, power outage, software error, hardware error, a fire in the machine room, even sabotage.

# Failure Classification

- Transaction failure :
  - Logical errors: The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - System errors: The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.
- System crash: a power failure or other hardware or software failure causes the system to crash.
  - Fail-stop assumption: The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the 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 disk block loses its content as a result of either a head crash or failure during a data-transfer operation.

## Recovery algorithms

- To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data.
- Next, we must consider how these failure modes affect the contents of the database.
- We can then propose algorithms to ensure database consistency and transaction atomicity despite failures.

## Recovery algorithms

- Recovery algorithms, have two parts:

1. Precondition: Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Postcondition: Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability

- The recovery scheme must provide high availability; that is, it must minimize the time for which the database is not usable after a failure.

## Recovery and Atomicity

- When a DBMS recovers from a crash, it should maintain the following −
  - It should check the states of all the transactions, which were being executed.
  - A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
  - It should check whether the transaction can be completed now or it needs to be rolled back.
  - No transactions would be allowed to leave the DBMS in an inconsistent state

## Recovery and Atomicity

- There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −
  - Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
  - Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

## Log-based Recovery

- The most widely used structure for recording db modification is the log.
- Log is a sequence of records
  - Which maintains the records of actions performed by a transaction.
    - It contains information about the start and end of each transaction and any updates which occur in the **transaction**.
    - The log keeps track of all transaction operations that affect the values of database items.
  - It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

## Log-based recovery

- Log-based recovery works as follows –
  - The log file is kept on a stable storage media.
  - When a transaction enters the system and starts execution, it writes a log about it.
  - At the time of a system crash, item is searched back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process.

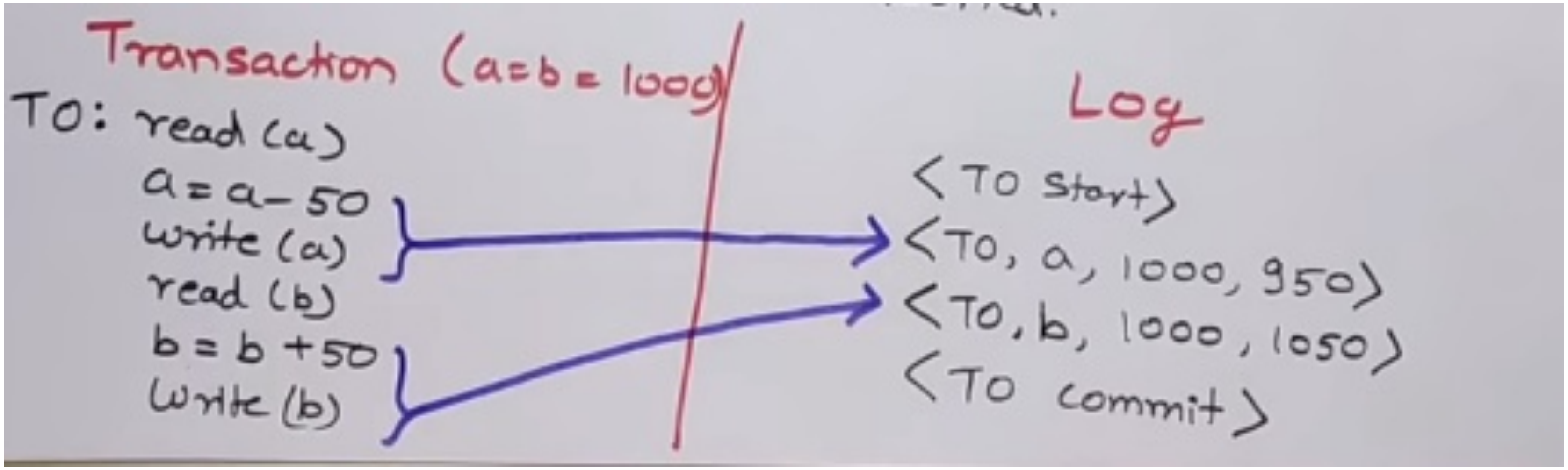
## Log-based recovery: Log update fields

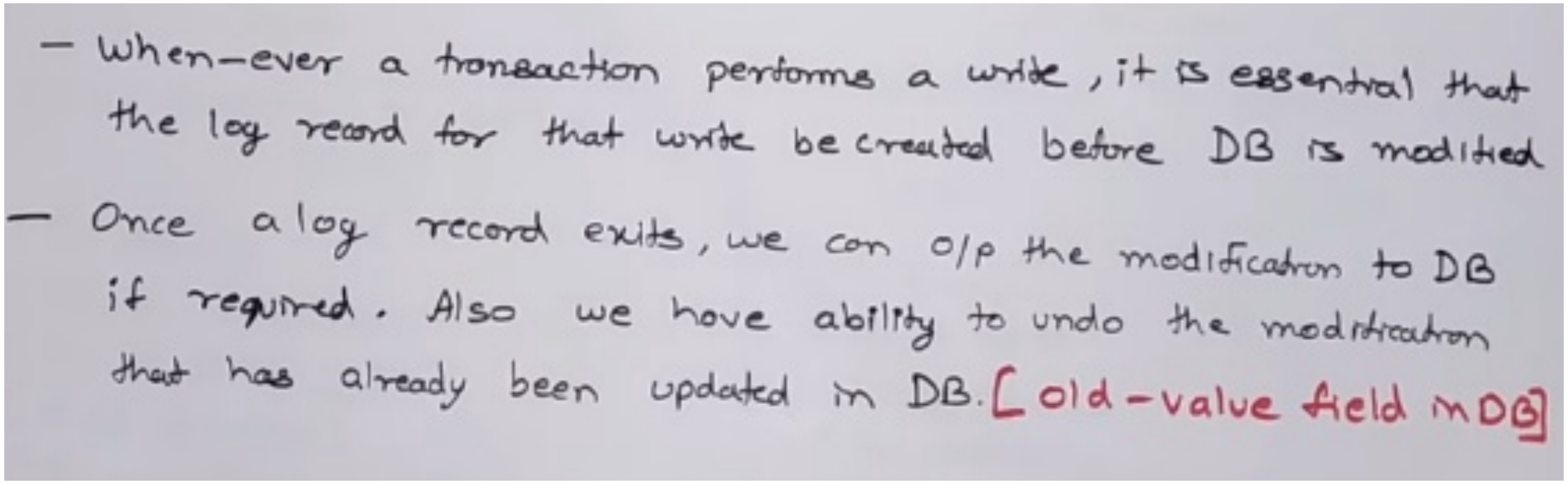
- Transaction identifier
  - which is the unique identifier of the transaction that performed the write operation.
- Data-item identifier
  - which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- Old value
  - which is the value of the data item prior to the write.
- New value
  - which is the value that the data item will have after the write.

# Log-based recovery: Log update fields

- start_transaction($T_i$): This log entry records that transaction T starts the execution.
- read_item($T_i$, $X_i$): This log entry records that transaction T reads the value of database item X
- write_item($T_i$, $X_i$, old_value, new_value): This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a beforeimage of X, and the new value is known as an afterimage of X.
- commit($T_i$): This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- abort($T_i$): This records that transaction T has been aborted.

# Log-based recovery: Log update fields





- When-ever a transaction performs a write, it is essential that the log record for that write be created before DB is modified
- Once a log record exits, we can o/p the modification to DB if required. Also we have ability to undo the modification that has already been updated in DB. [old-value field in DB]

# Log-based recovery: database update strategies

**The database can be modified using two approaches −**

- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- No Undo/redo algorithm

- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.
- DB is **Immediately update** by the transaction operation during the execution of transaction even before it **reaches commit operation.**
- **If abort/failure occur before** transaction **reaches commit, a rollback or undo operation** need to be done to restore db to last consistent state.
- Undo/redo algorithm

## Immediate

(modification in active state)

<T0 Start>
<T0, A, 1000, 950>
<T0, B, 2000, 2050>
<T0 commit>
<T1 Start>
<T1, C, 700, 600>
<T1 commit>

| LOG | DB |
|---|---|
| <T0 start> | |
| <T0, A, 1000, 950> | |
| <T0, B, 2000, 2050> | |
| — — — — — | [ A = 950 B = 2050 |
| <T0 Commit> | |
| <T1 Start> | |
| <T1, C, 700, 600> | |
| — — — — — | [ C = 600 |
| <T1 commit> | |

## Deffered

(modification in commit state)

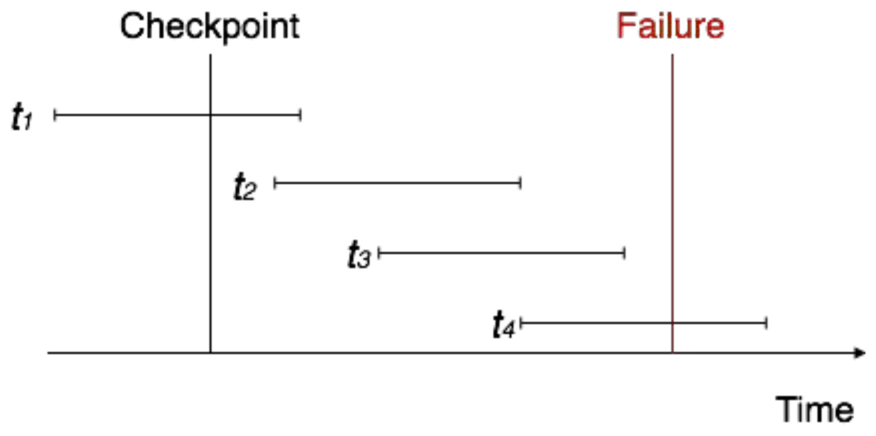
| LOG | DB |
|---|---|
| <T0 Start> | |
| <T0, A, 950> | |
| <T0, B, 2050> | |
| <T0, commit> | |
| {T0 Ends} —— | [ A = 950 B = 2050 ] |
| <T1 Starts> | |
| <T1, C, 600> | |
| <T1 commit> | |
| {T1 Ends} —— | [ C = 600 ] |

## Recovery with Concurrent Transactions

- When more than one transaction are being executed in parallel, the logs are interleaved.
- At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering.
- To ease this situation, most modern DBMS use the concept of 'checkpoints'.
- Checkpoint: Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.
-

## Recovery with Concurrent Transactions

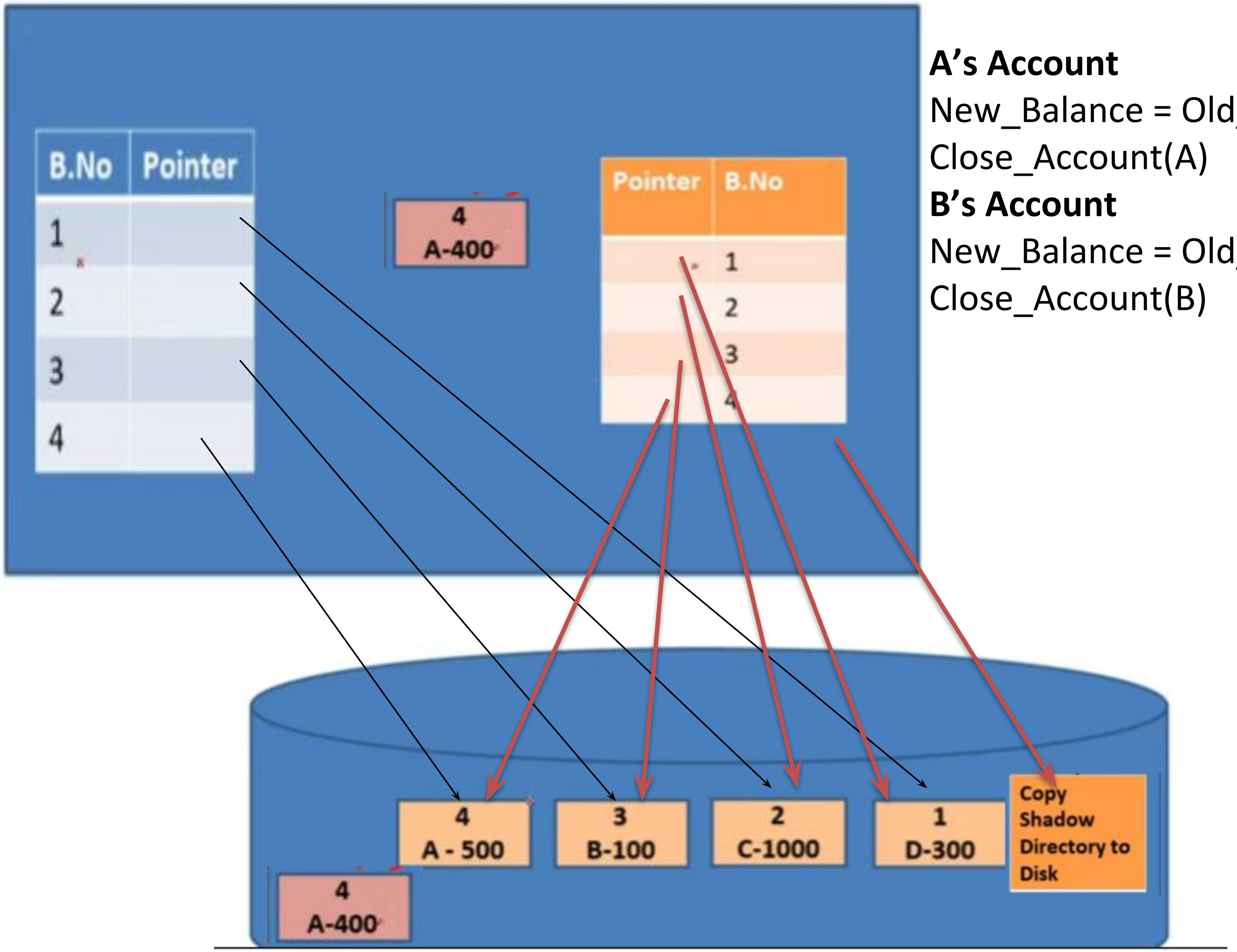
- When a system with concurrent transactions crashes and recovers, it behaves in the following manner –



# Recovery with Concurrent Transactions

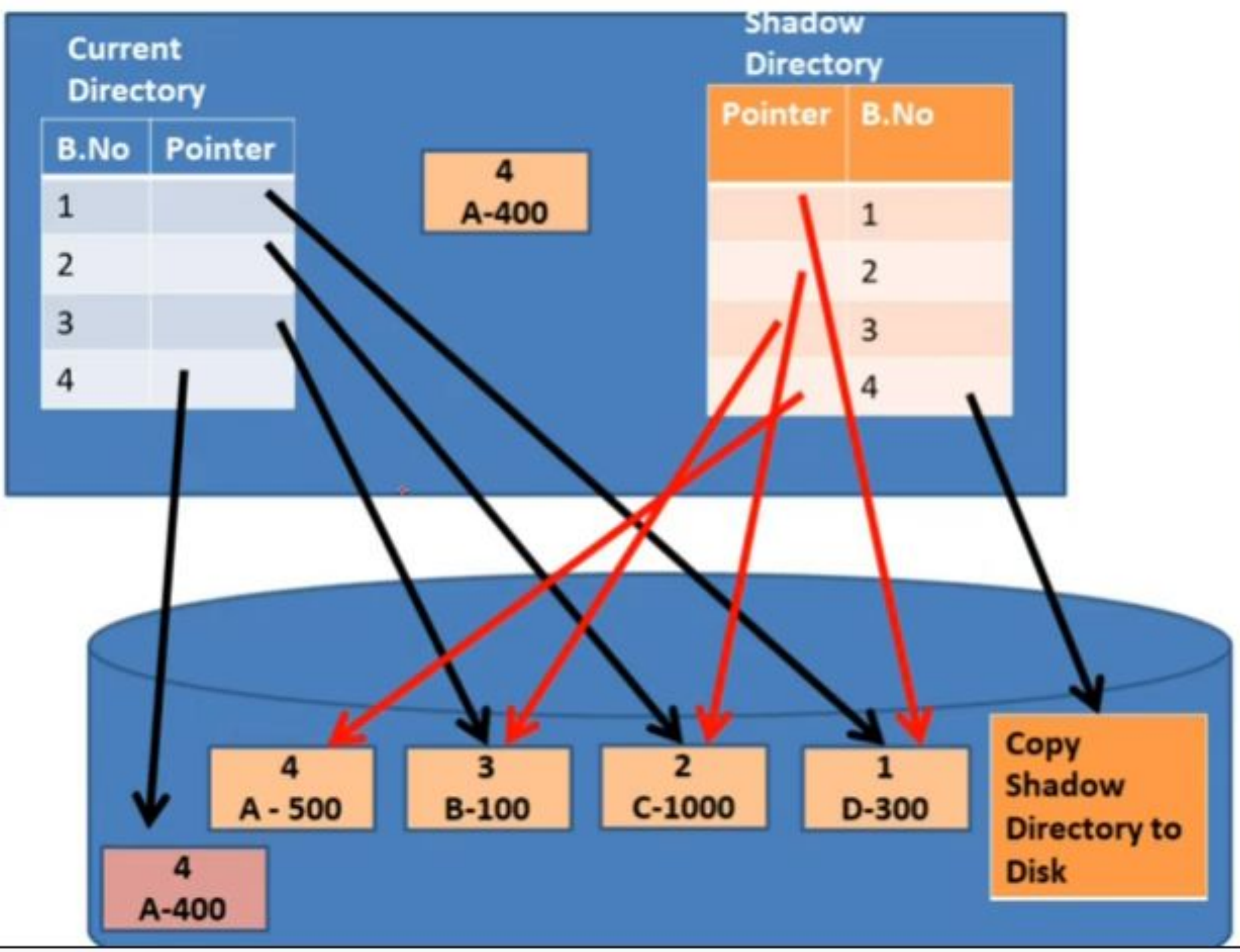
- The recovery system reads the logs backwards from the end to the last checkpoint.
  - It maintains two lists, an undo-list and a redo-list.
  - If the recovery system sees a log with <$T_n$, Start> and <$T_n$, Commit> or just <$T_n$, Commit>, it puts the transaction in the redo-list.
  - If the recovery system sees a log with <$T_n$, Start> but no commit or abort log found, it puts the transaction in undo-list.
  - All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

# Shadow Paging (No Undo and No Redo)

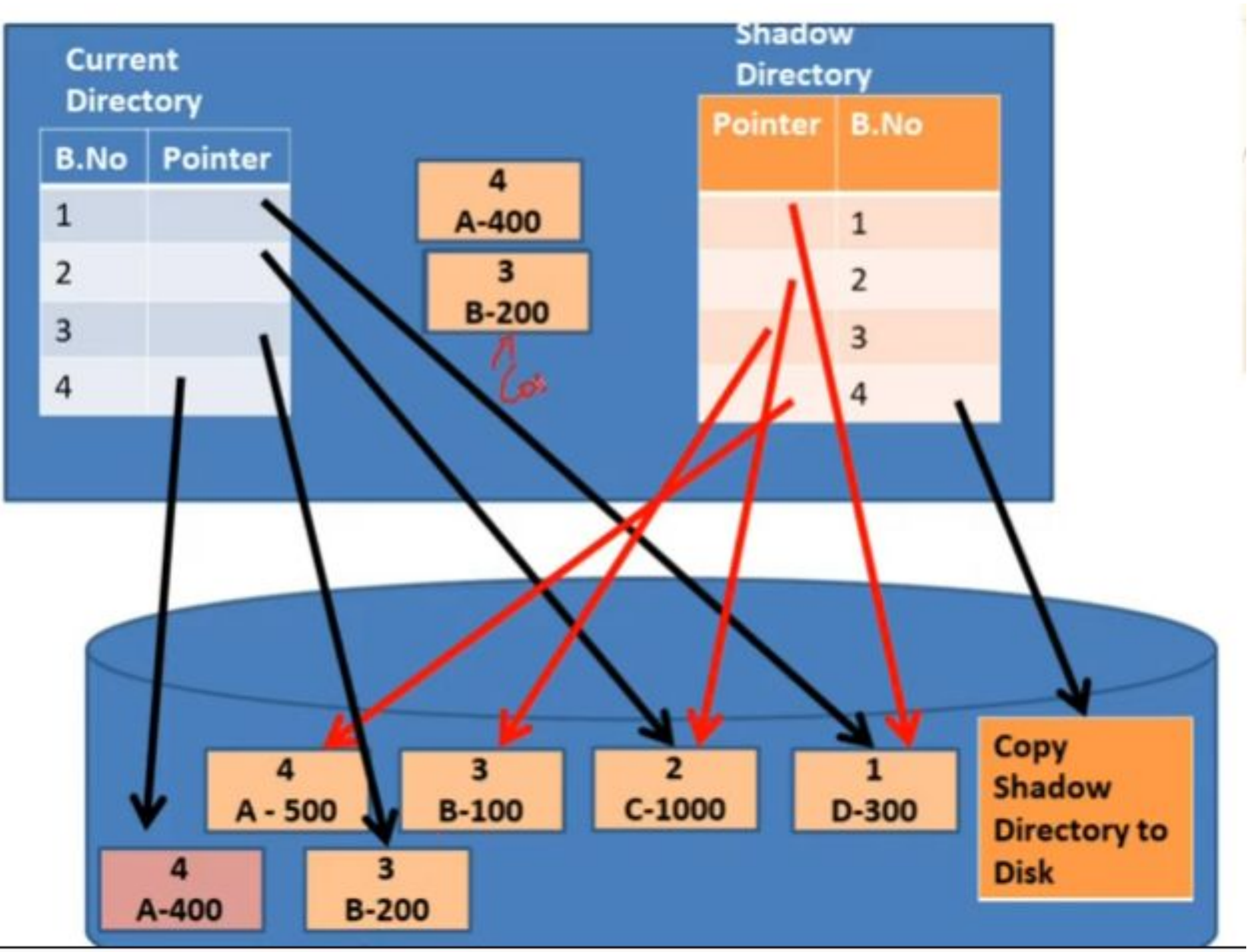


**A's Account**
New_Balance = Old_Balance – 100
Close_Account(A)
**B's Account**
New_Balance = Old_Balance + 100
Close_Account(B)

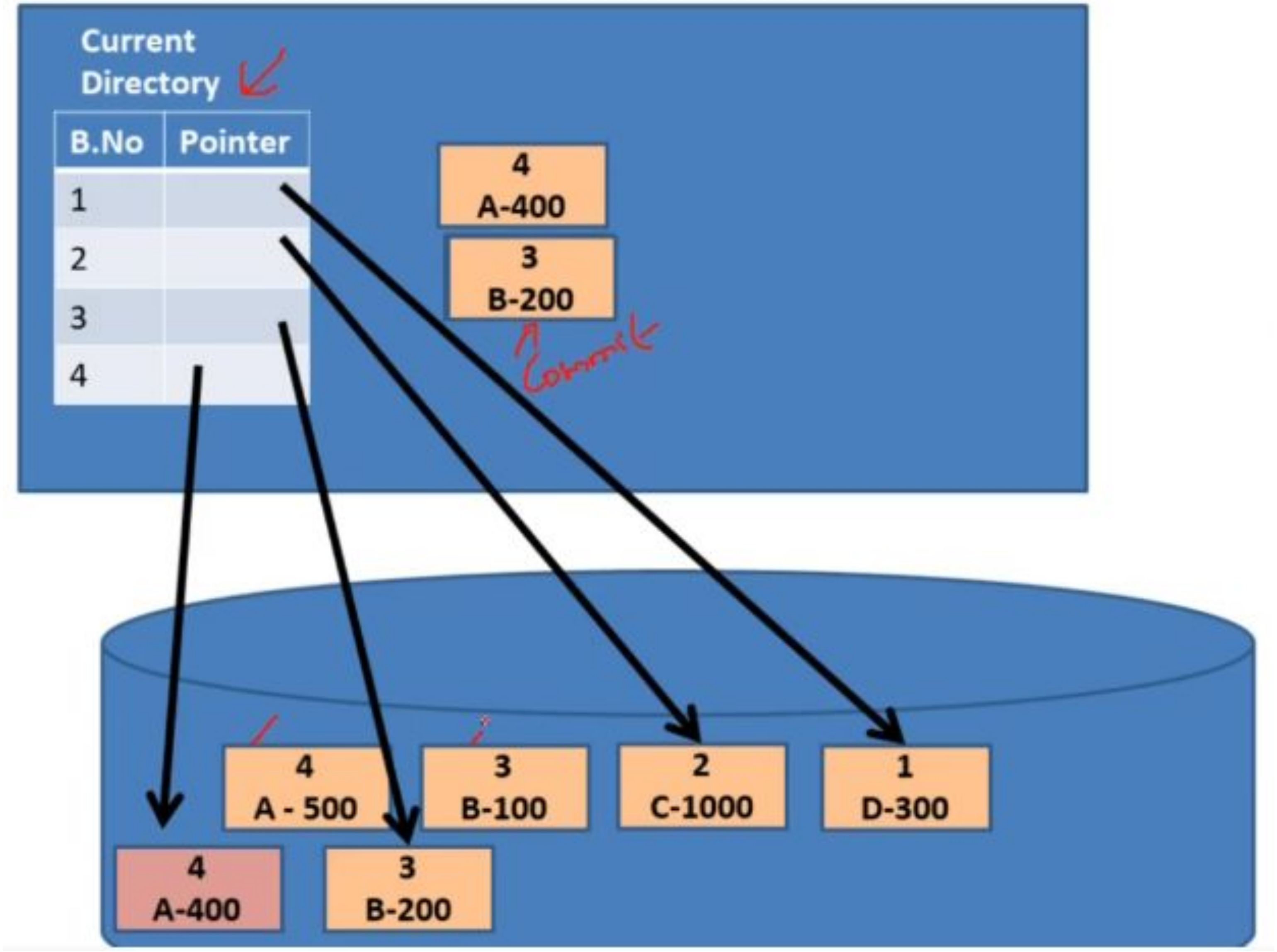
# Shadow Paging (No Undo and No Redo)



# Shadow Paging (No Undo and No Redo)



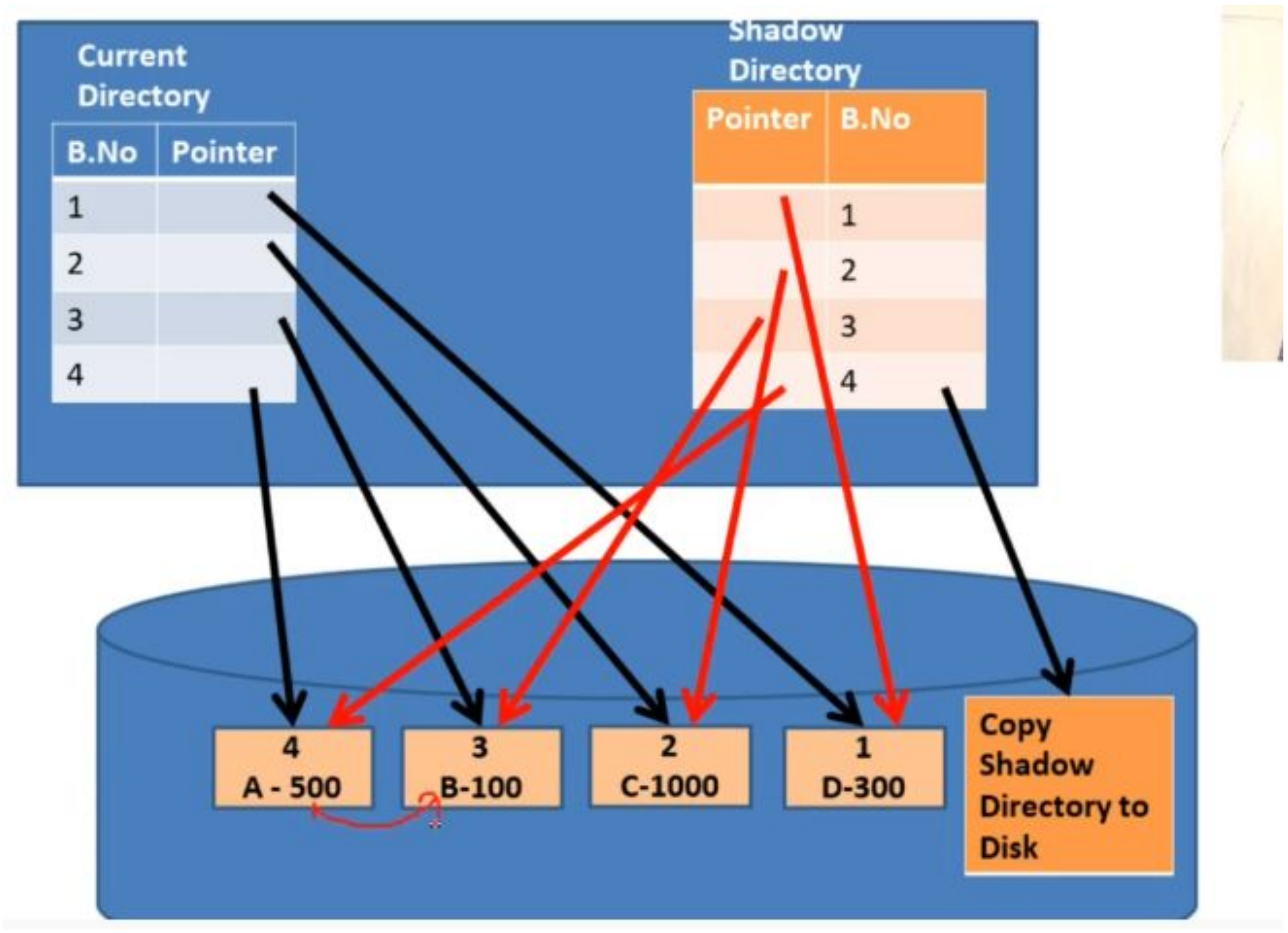
# Shadow Paging (No Undo and No Redo)



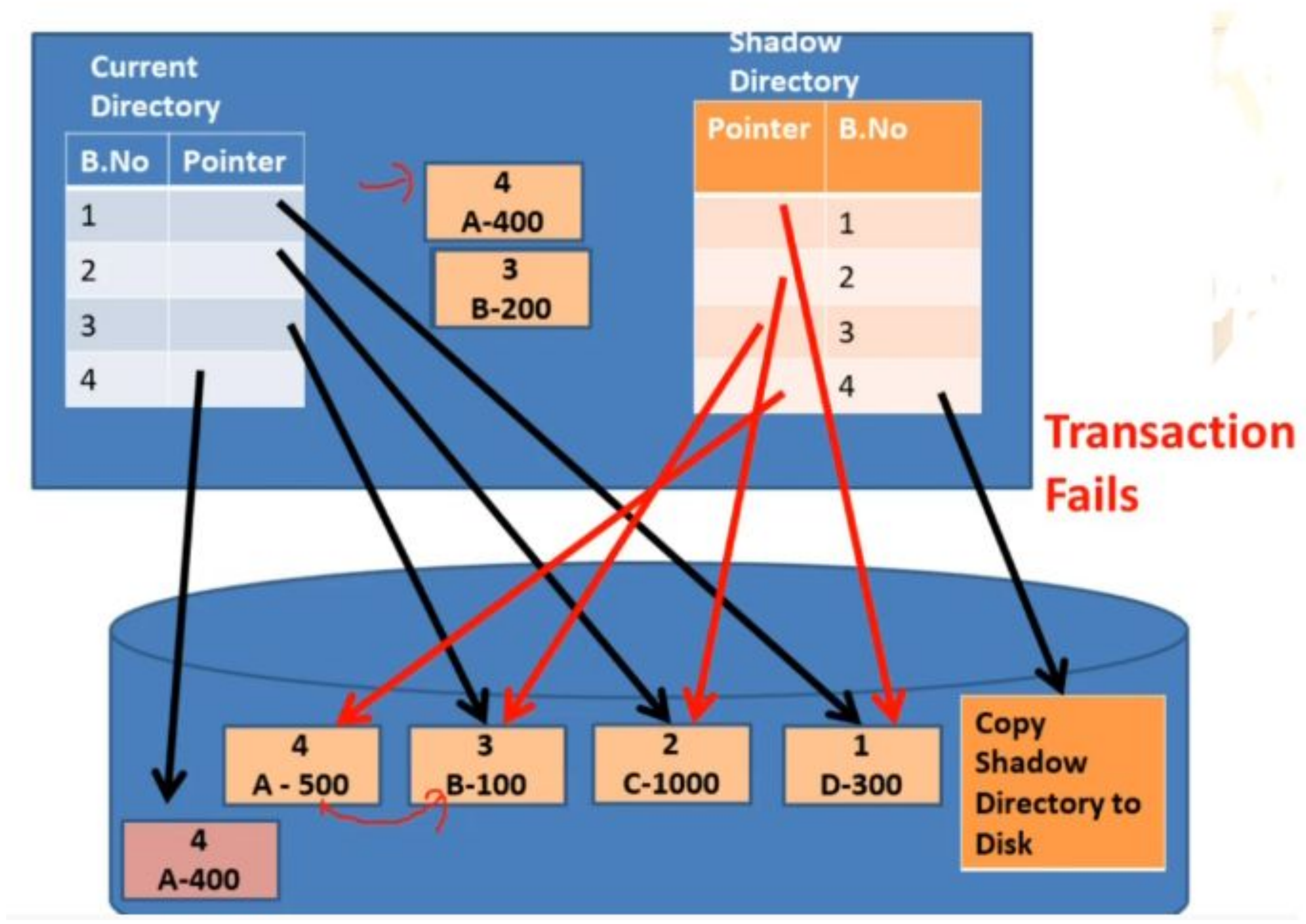
# Shadow Paging (No Undo and No Redo)

- **copy**-on-write technique.
- A  directory is getting created
  - It contains block no and pointer to each block
- Create the shadow of the directory
  - Copy in the disk
- Start the transaction
  - Transfer block A to main memory
  - Then reduce 100 there
  - write that back to the disks to new location
  - Current directory now point to new location
  - Transfer block B to main memory
  - Then add100 there
  - write that back to the disks to new location
  - Current directory now point to new location
- If Transaction commit
  - Discard shadow directory
  - The blocks which are not pointed by current directory are de-allocated
- If Transaction fails
  - Discard current directory

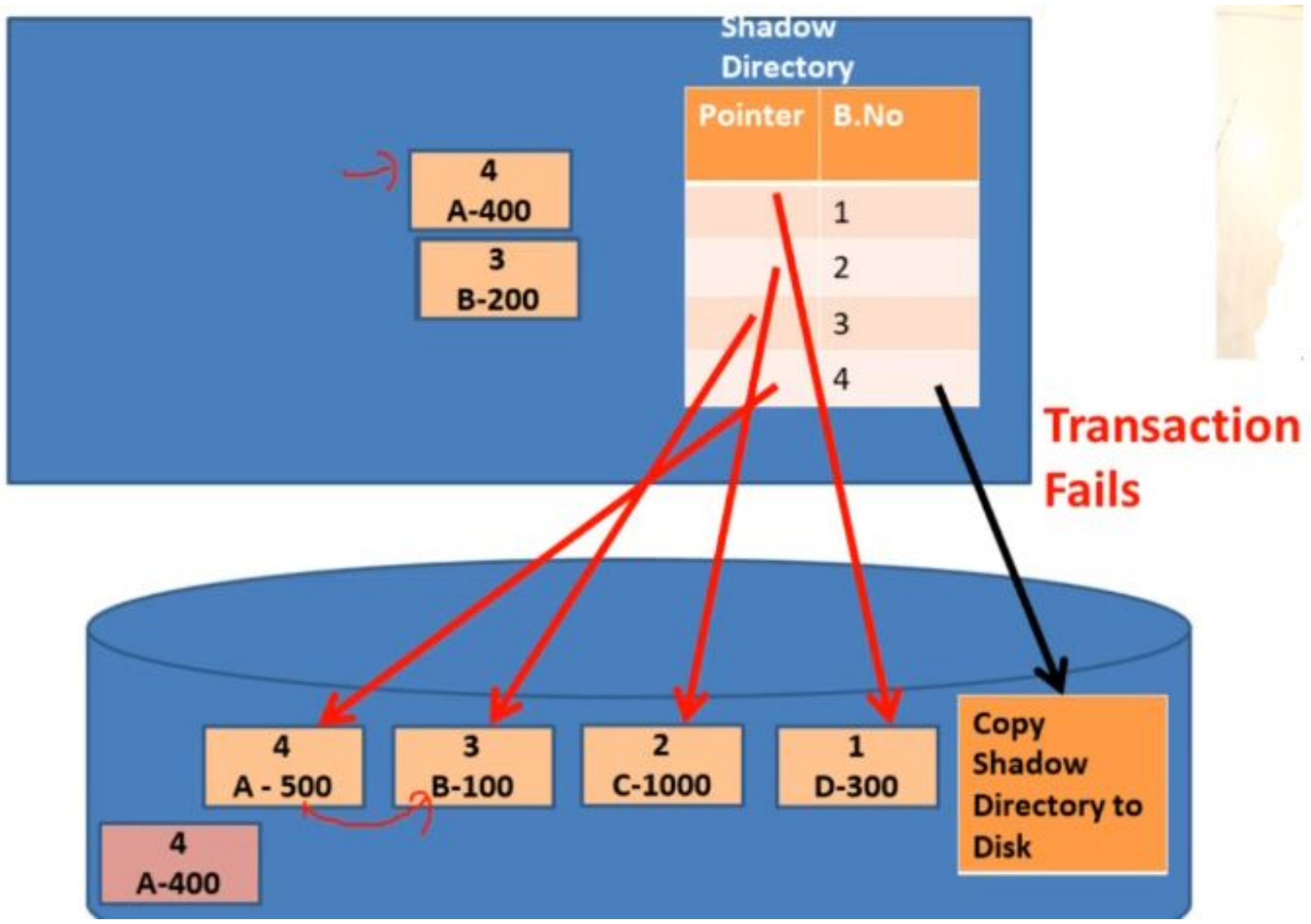
# Shadow Paging (No Undo and No Redo)



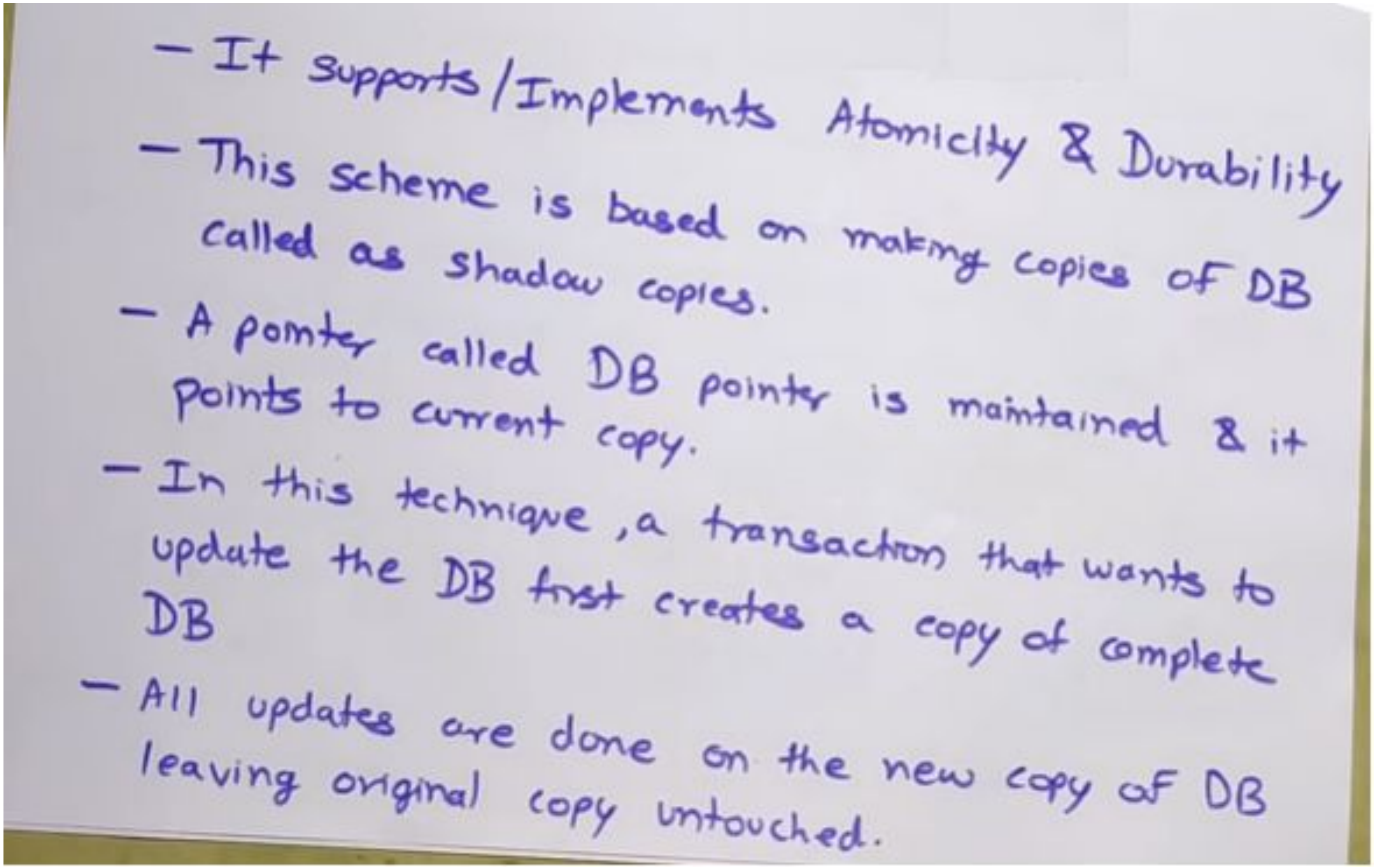
# Shadow Paging (No Undo and No Redo)



# Shadow Paging (No Undo and No Redo)



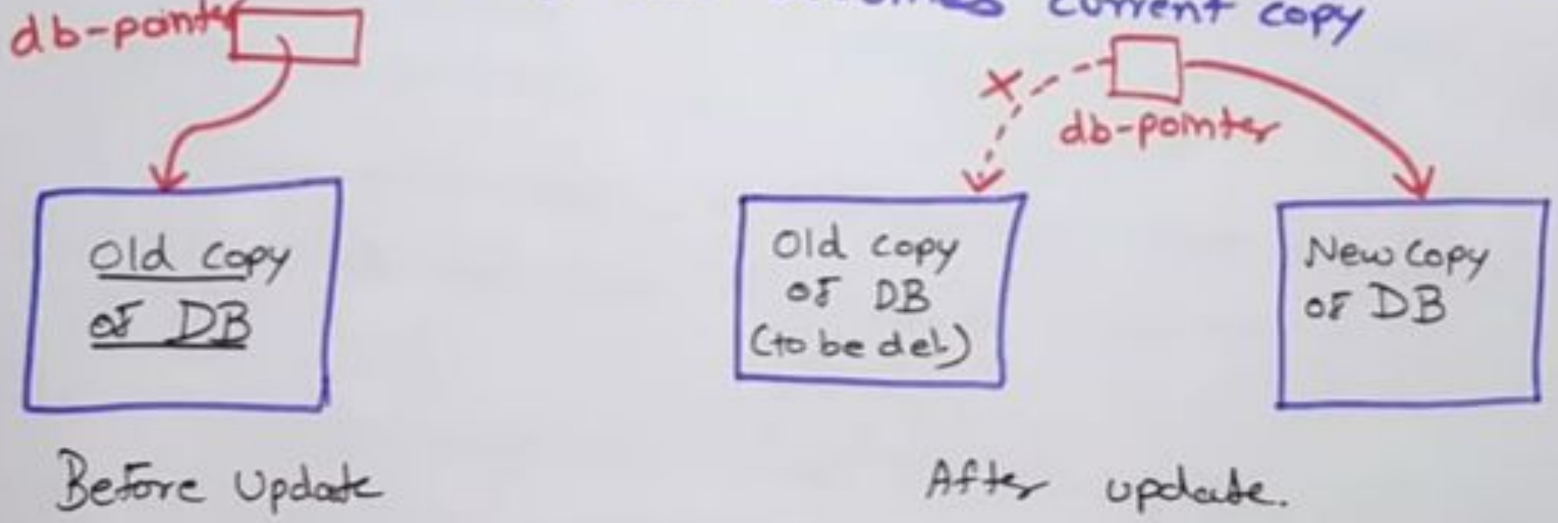
# Shadow Copy



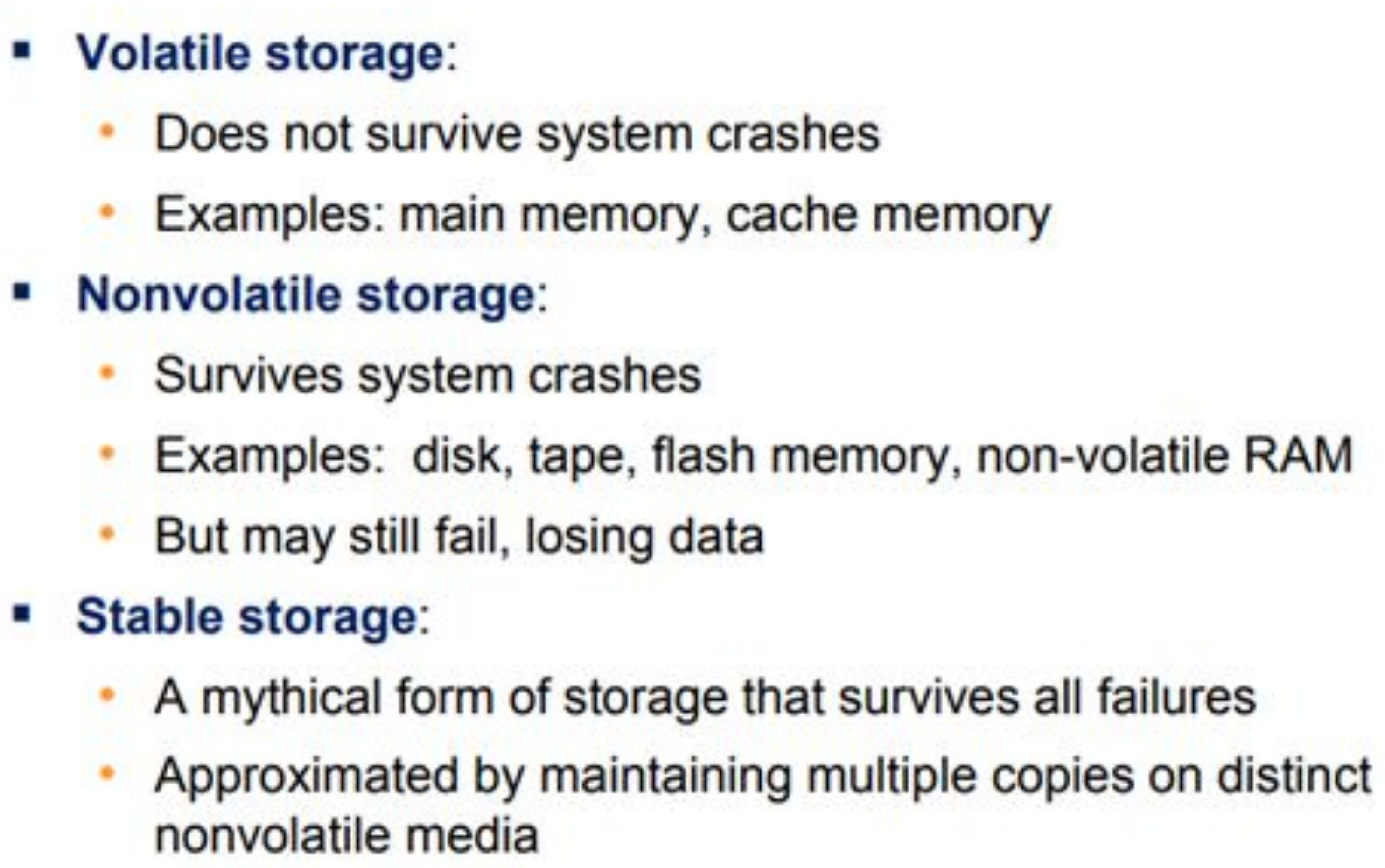
- It supports/Implements Atomicity & Durability
- This scheme is based on making copies of DB called as shadow copies.
- A pointer called DB pointer is maintained & it points to current copy.
- In this technique, a transaction that wants to update the DB first creates a copy of complete DB
- All updates are done on the new copy of DB leaving original copy untouched.

# Shadow Copy

- If transaction completes, it is committed as follows
  - OS is asked to make sure all the updates of new copy of DB are written out to the disk.
- After OS has written successfully to disk, the db-pointer is updated ~~ta~~ and points to the new copy of DB; and this copy now becomes current copy



# 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 RAM
  - But may still fail, losing data
- **Stable storage**:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media

## 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 nonvolatile 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

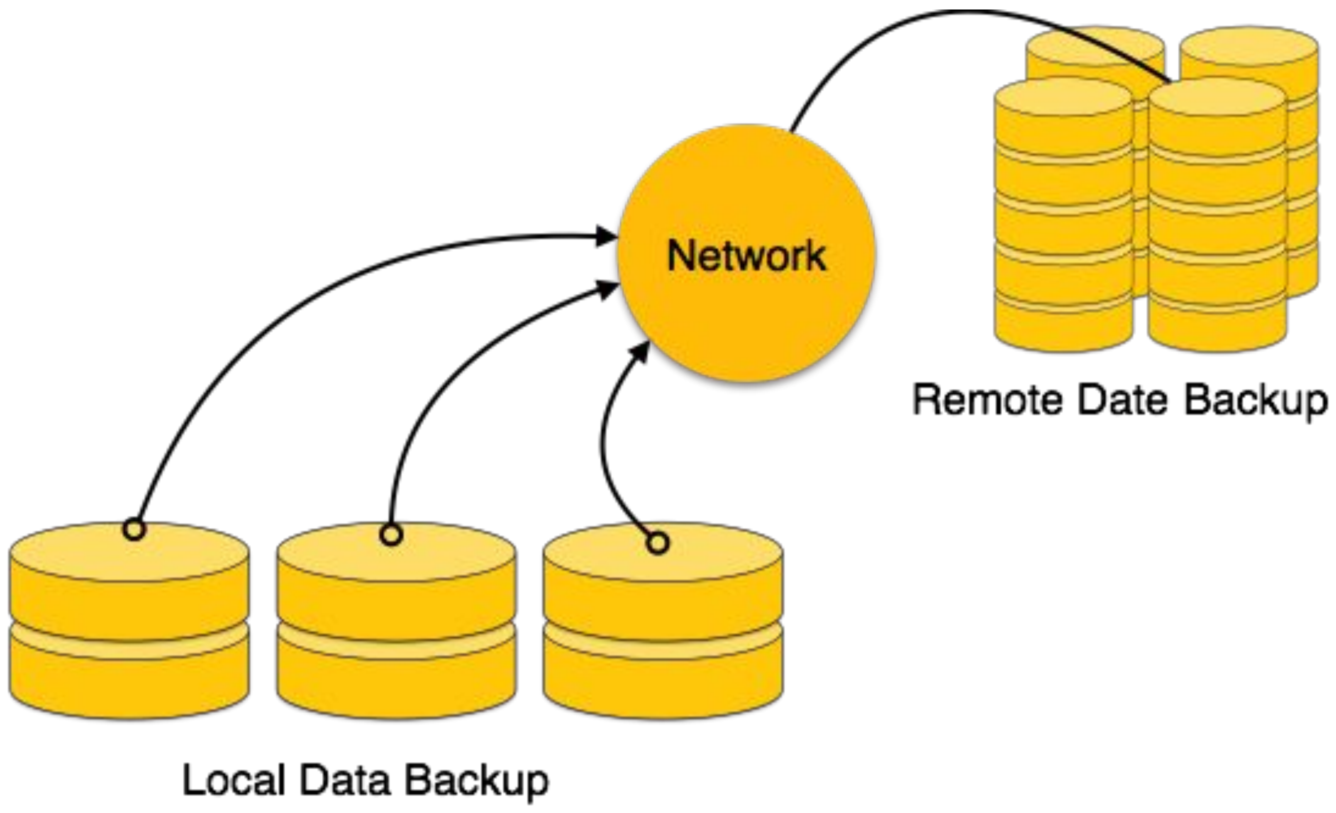
## Failure with Loss of Nonvolatile Storage

- To recover from disk failure
  - restore database from most recent dump
  - Consult the log and redo all transactions that committed after the dump

## Remote Backup

- Remote backup provides a sense of security in case the primary location where the database is located gets destroyed.
- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.
- Remote backup can be offline or real-time or online. In case it is offline, it is maintained manually.

## Remote Backup



## Remote Backup

- Online backup systems are more real-time and lifesavers for database administrators and investors.
  - An online backup system is a mechanism where every bit of the real-time data is backed up simultaneously at two distant places.
  - One of them is directly connected to the system and the other one is kept at a remote place as backup.
- As soon as the primary database storage fails, the backup system senses the failure and switches the user system to the remote storage
  - Sometimes this is so instant that the users can’t even realize a failure.

## Advance recovery Algorithm

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

## Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) is based on Three main principle
  - Write-ahead logging
  - Repeating history during Redo
    - On restart after a crash, ARIES retraces the actions of a database before the crash and brings the system back to the exact state that it was in before the crash. Then it undoes the transactions still active at crash time.
  - Logging changes during Undo
    - Changes made to the database while undoing transactions are logged to ensure such an action isn't repeated in the event of repeated restarts.

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- Every update operation writes a log- record which is one of the following :
  - **Undo-only log record**
    - restore the old value
  - **Redo-only log record**
    - Update the new values
  - **Undo-Redo log record**

- Write Ahead Log strategy
  - Log is written before any update is made to the database
    - Transaction is not allowed to modify the physical database until the undo portion is written
    - log must be written to stable storage before changes to the database are written to disk.
- Once the logs are recorded on stable storage, physical copy of db can be modify.

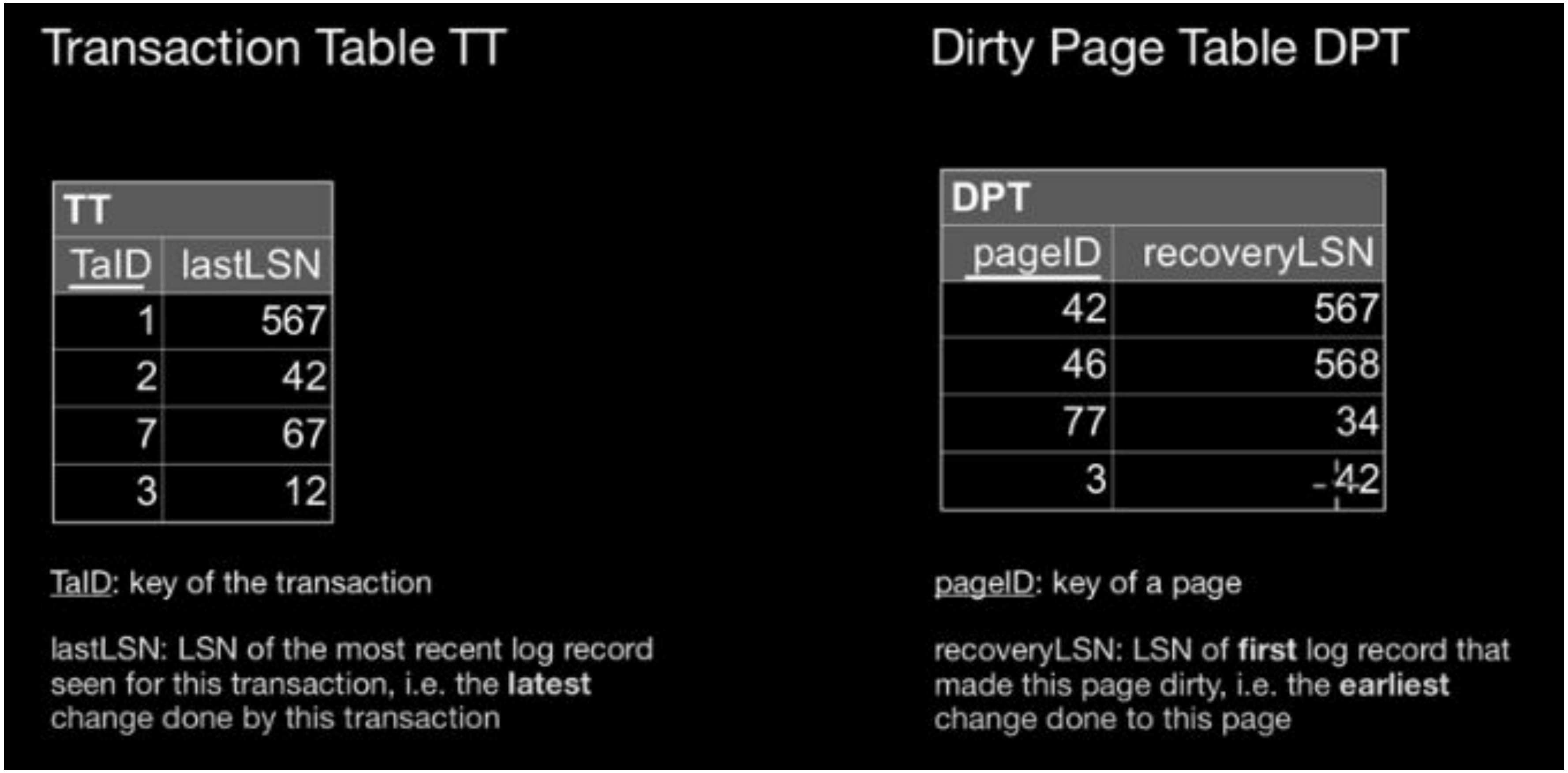
# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- For the ARIES algorithm to work a number of log records have to be created during the operation of the database.
  - Every log record is assigned a unique and monotonically increasing log sequence number (LSN).
  - Every data page has a pageLSN field that is set to the LSN of the log record corresponding to the last update on the page.
- WAL requires that the log record corresponding to an update make it to stable storage before the data page corresponding to that update is written to disk.
  - A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full.
  - For performance reasons, each log write is not immediately forced to disk

## Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- To gather the necessary information for the logging two data structures have to be maintained: the dirty page table (DPT) and the transaction table (TT).
  - The dirty page table keeps record of all the pages that have been modified and not yet written back to disc and the first Sequence Number that caused that page to become dirty.
  - The transaction table contains all transactions that are currently running and the Sequence Number of the last log entry they caused.

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



Transaction Table TT

| TT | |
|---|---|
| TaID | lastLSN |
| 1 | 567 |
| 2 | 42 |
| 7 | 67 |
| 3 | 12 |

TaID: key of the transaction

lastLSN: LSN of the most recent log record seen for this transaction, i.e. the **latest** change done by this transaction

Dirty Page Table DPT

| DPT | |
|---|---|
| pageID | recoveryLSN |
| 42 | 567 |
| 46 | 568 |
| 77 | 34 |
| 3 | 42 |

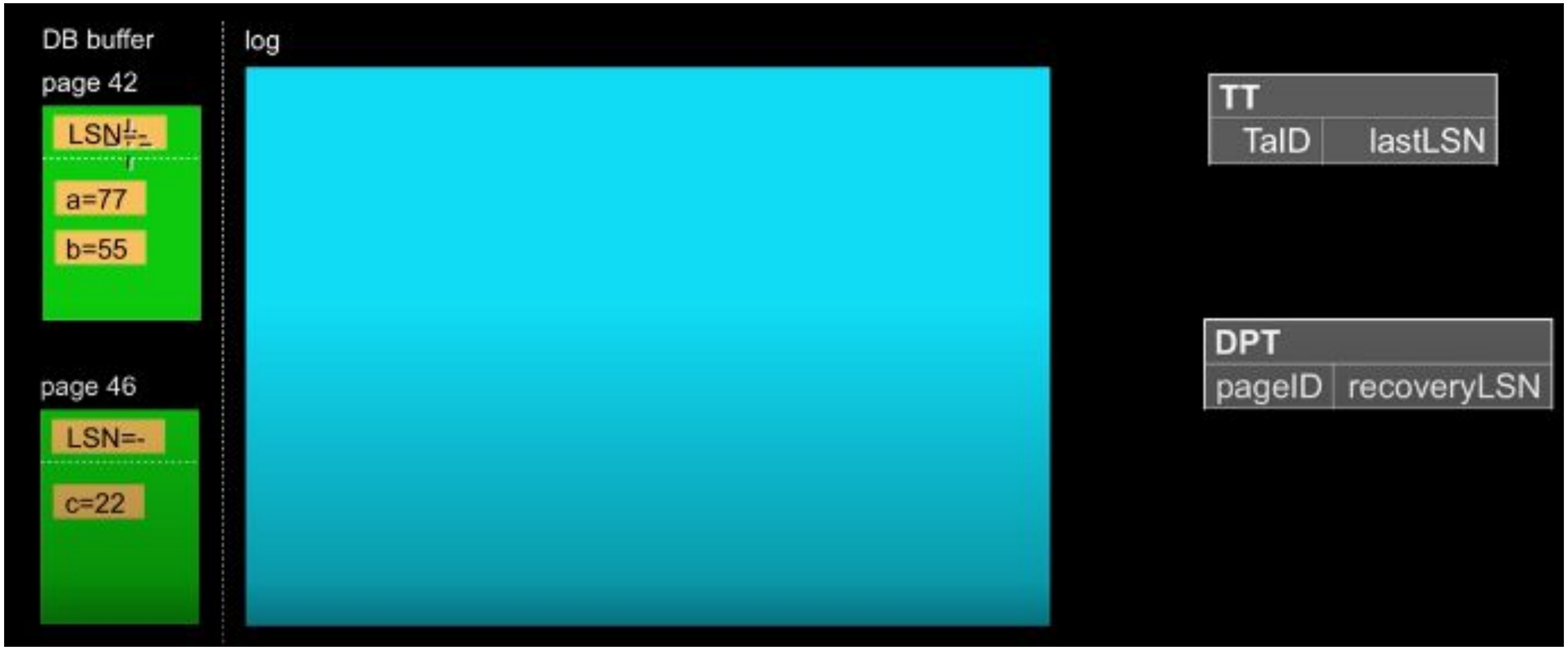
pageID: key of a page

recoveryLSN: LSN of **first** log record that made this page dirty, i.e. the **earliest** change done to this page

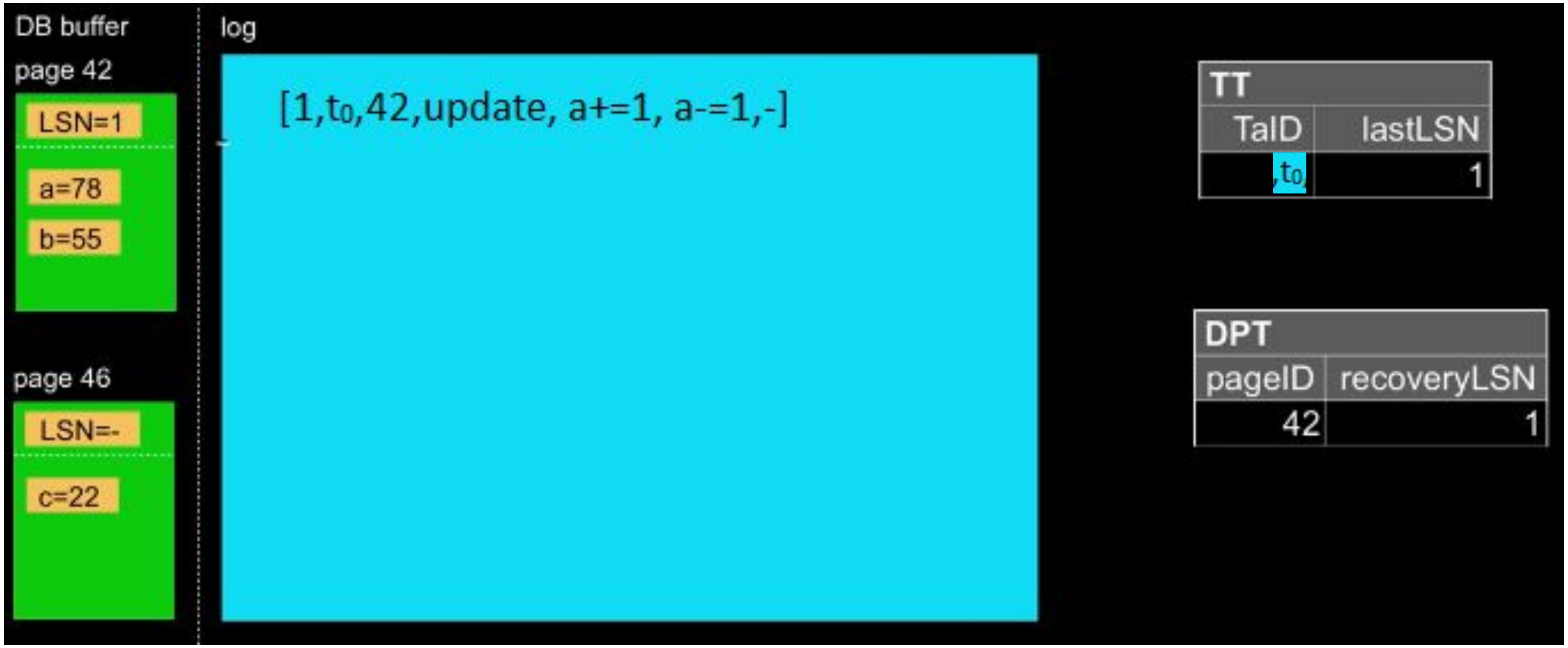
# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- Log format
  - (Sequence Number, Transaction ID, Page ID, Type,  Redoinfo, Undoinfo, Previous Sequence Number).
- Redoinfo: information how to redo the change reflected by this log record
- Undoinfo: information how to Undo the change reflected by this log record
- The Previous Sequence Number is a reference to the previous log record that was created for this transaction.
  - In the case of an aborted transaction, it's possible to traverse the log file in reverse order using the Previous Sequence Numbers, undoing all actions taken within the specific transaction.
- Type
  - kind of log record; update, commit, compensation

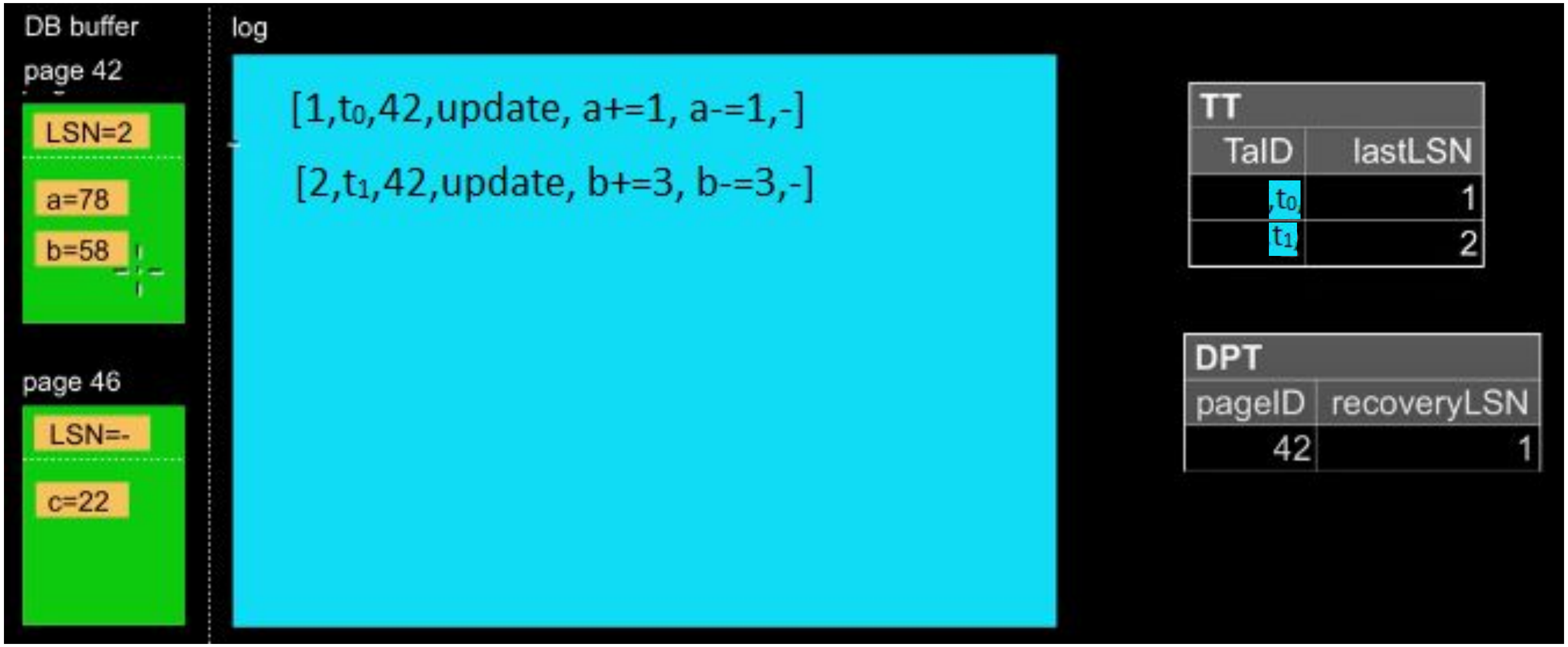
# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



DB buffer

page 42

LSN=2

a=78

b=58

page 46

LSN=-

c=22

log

```
[1,t0,42,update, a+=1, a-=1,-]
[2,t1,42,update, b+=3, b-=3,-]
```

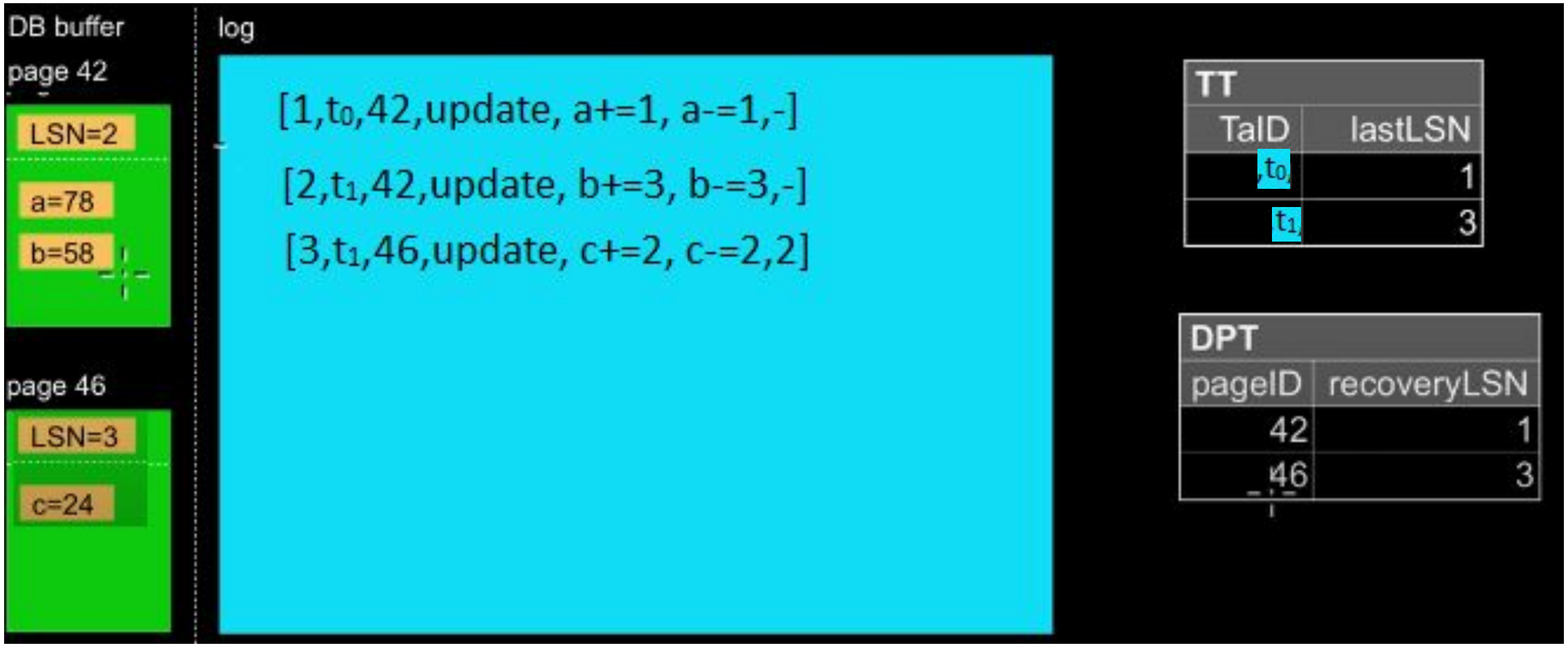
TT

| TaID | lastLSN |
|---|---|
| $t_0$ | 1 |
| $t_1$ | 2 |

DPT

| pageID | recoveryLSN |
|---|---|
| 42 | 1 |

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



[1,$t_0$,42,update, a+=1, a-=1,-]

[2,$t_1$,42,update, b+=3, b-=3,-]

[3,$t_1$,46,update, c+=2, c-=2,2]

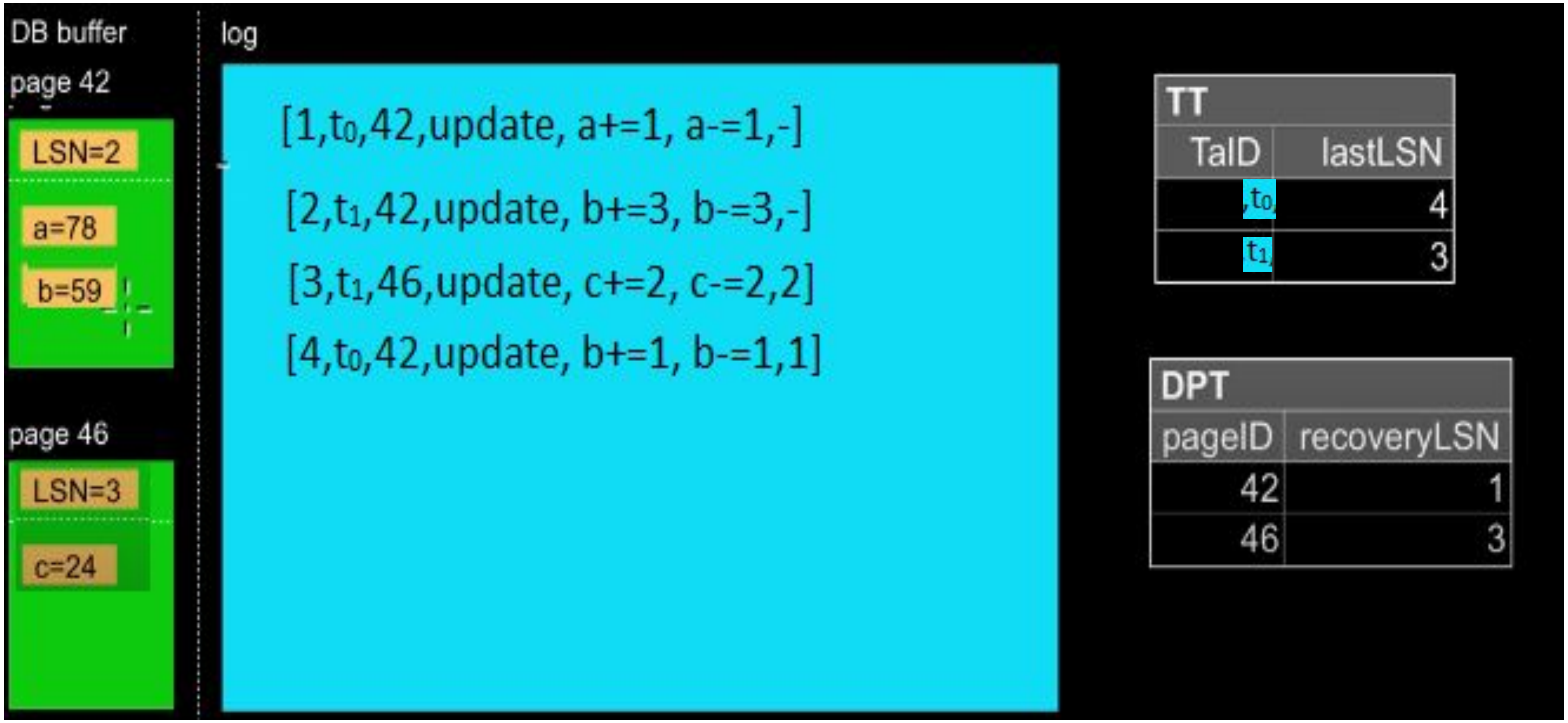
TT

| TaID | lastLSN |
|---|---|
| $t_0$ | 1 |
| $t_1$ | 3 |

DPT

| pageID | recoveryLSN |
|---|---|
| 42 | 1 |
| 46 | 3 |

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



```
[1,t0,42,update, a+=1, a-=1,-]
[2,t1,42,update, b+=3, b-=3,-]
[3,t1,46,update, c+=2, c-=2,2]
[4,t0,42,update, b+=1, b-=1,1]
```

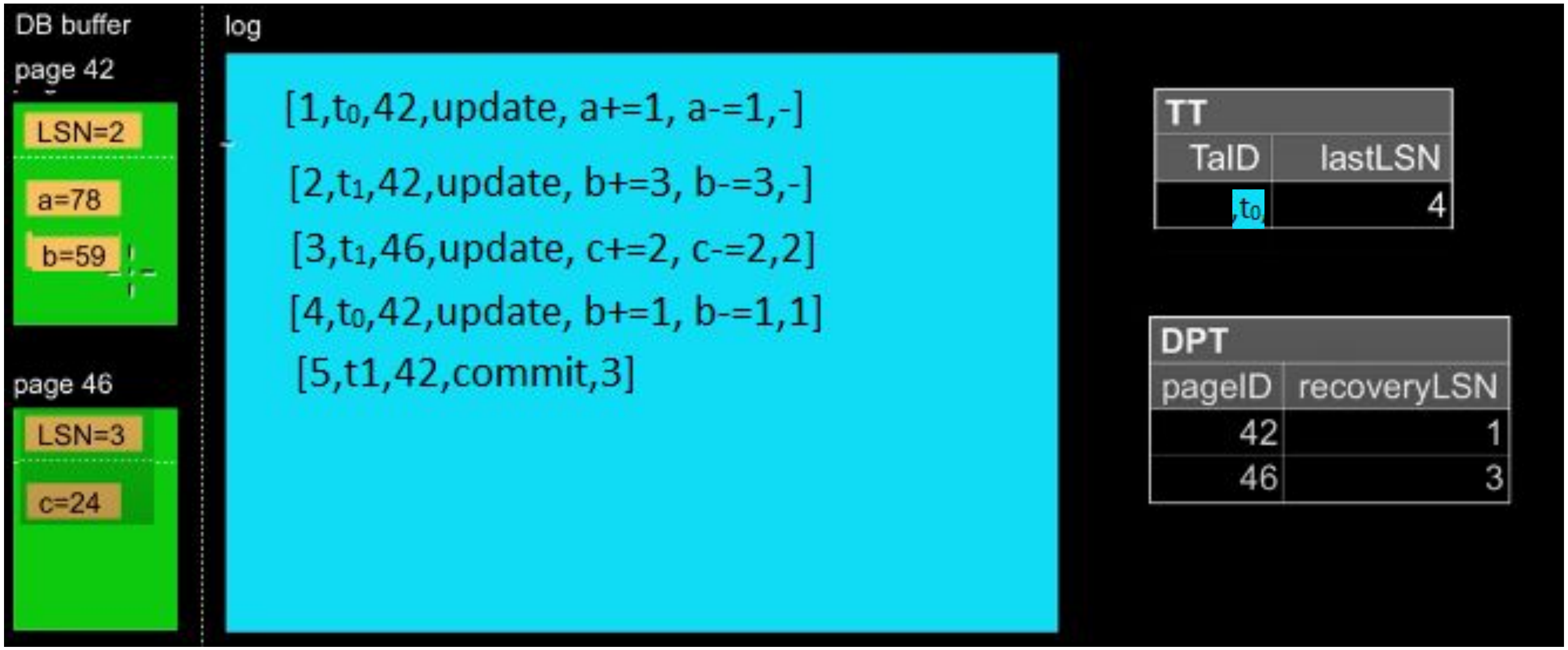
TT

| TaID | lastLSN |
|---|---|
| $t_0$ | 4 |
| $t_1$ | 3 |

DPT

| pageID | recoveryLSN |
|---|---|
| 42 | 1 |
| 46 | 3 |

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



[1,t0,42,update, a+=1, a-=1,-]
[2,t1,42,update, b+=3, b-=3,-]
[3,t1,46,update, c+=2, c-=2,2]
[4,t0,42,update, b+=1, b-=1,1]
[5,t1,42,commit,3]

TT

| TaID | lastLSN |
|---|---|
| t0 | 4 |

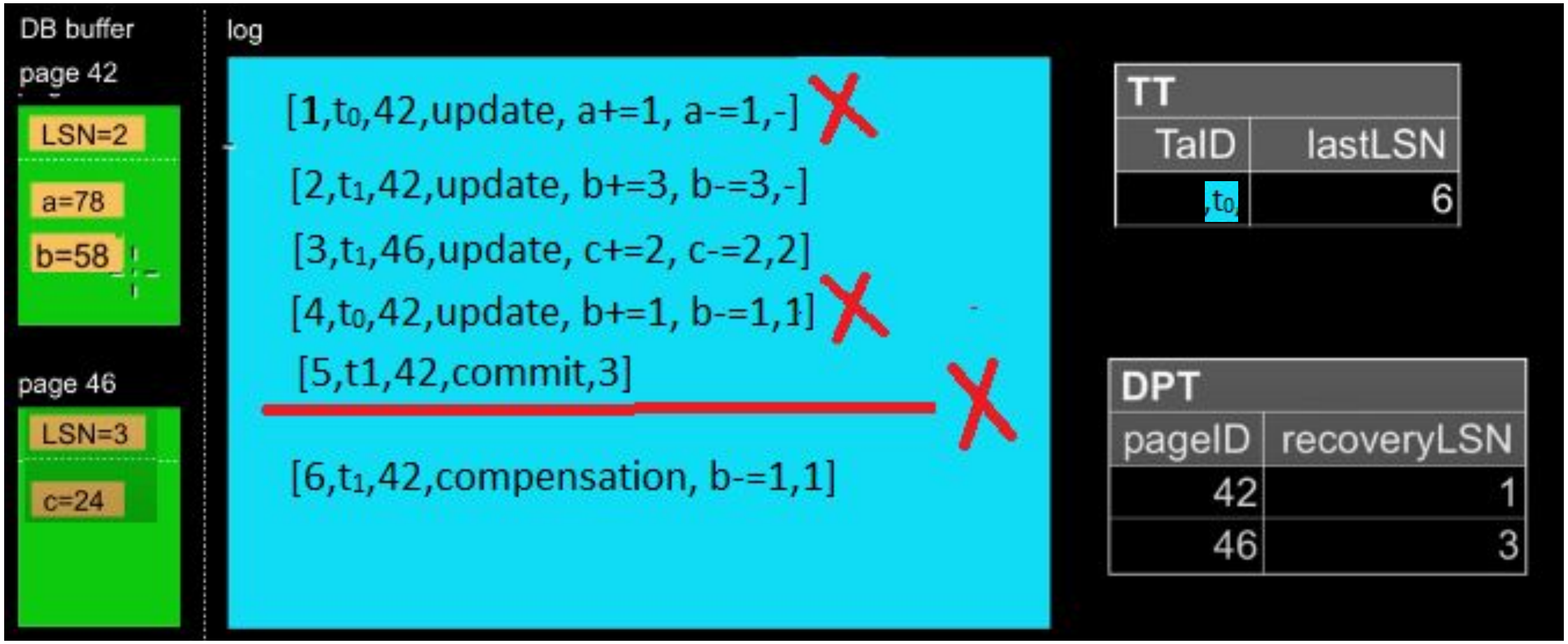
DPT

| pageID | recoveryLSN |
|---|---|
| 42 | 1 |
| 46 | 3 |

## Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- Compensation log format
  - (Sequence Number, Transaction ID, Page ID, type,Redoinfo, Previous Sequence Number, Next Undo Sequence Number)

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



[1,$t_0$,42,update, a+=1, a-=1,-]
[2,$t_1$,42,update, b+=3, b-=3,-]
[3,$t_1$,46,update, c+=2, c-=2,2]
[4,$t_0$,42,update, b+=1, b-=1,1]
[5,t1,42,commit,3]
[6,$t_1$,42,compensation, b-=1,1]

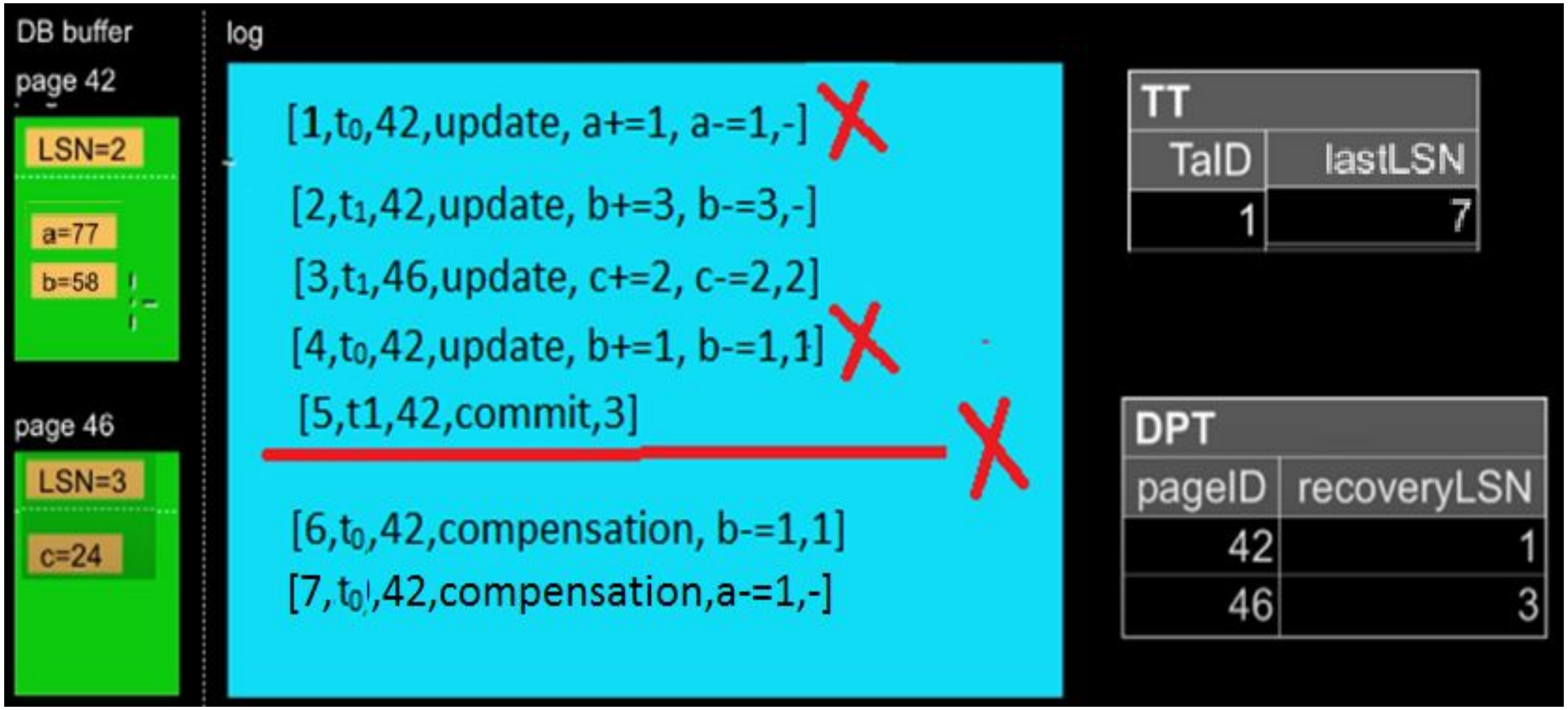
TT

| TaID | lastLSN |
|---|---|
| $t_0$ | 6 |

DPT

| pageID | recoveryLSN |
|---|---|
| 42 | 1 |
| 46 | 3 |

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)



[1,$t_0$,42,update, a+=1, a-=1,-]
[2,$t_1$,42,update, b+=3, b-=3,-]
[3,$t_1$,46,update, c+=2, c-=2,2]
[4,$t_0$,42,update, b+=1, b-=1,1]
[5,t1,42,commit,3]

[6,$t_0$,42,compensation, b-=1,1]
[7,$t_0$,42,compensation,a-=1,-]

**TT**

| TaID | lastLSN |
|---|---|
| 1 | 7 |

**DPT**

| pageID | recoveryLSN |
|---|---|
| 42 | 1 |
| 46 | 3 |

# Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)

- Recovery
  - The recovery works in three phases
  - Analysis phase,
    - computes all the necessary information from the logfile.
  - The Redo phase
    - restores the database to the exact state at the crash, including all the changes of uncommitted transactions that were running at that point in time.
  - The Undo phase
    - undoes all uncommitted changes, leaving the database in a consistent state.