CMPT 506

Database Recovery Techniques

Read Chapters 23

Dr. Abdelkarim Erradi

Department of Computer Science and Engineering

QU

Database Recovery

Back

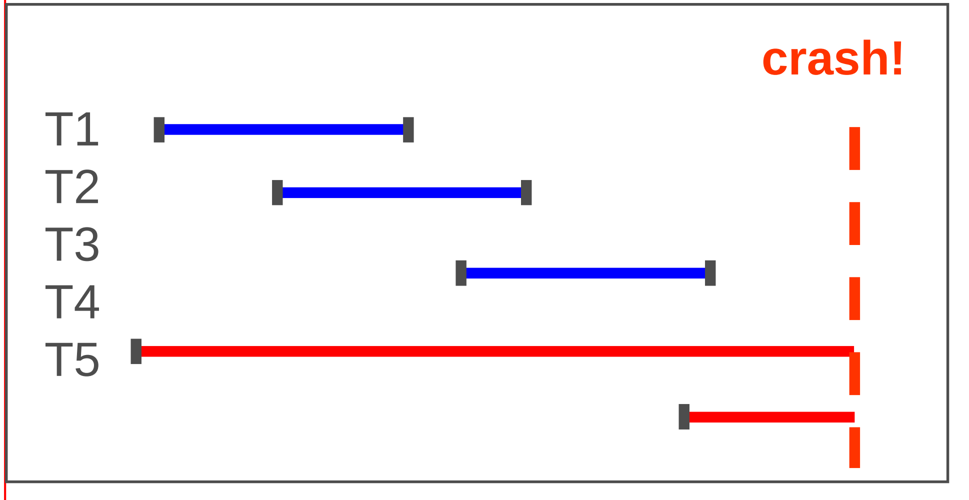
Motivation

- Purpose of database recovery is to ensure the transaction properties of:

 - **Atomicity:** Support the ability to rollback a transaction (via logging database changes i.e., updates, inserts and deletes)

- Durability: Bring the database into the last consistent state, which existed prior to the failure

- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable
  - T4 & T5 should be aborted (effects undone)



Types of Failure

1. Transaction failure: Transactions may fail because of incorrect input, deadlock, logical error (e.g., division by 0) etc.

2. System failure: System may fail because of addressing error, operating system fault, RAM failure, etc.

- For 1 & 2 Data on disk still there on restart

=> Solution: atomicity via logging

3. Media failure: Disk head crash, etc.

- Data on disks lost! => **Solution: Restore from backup**

4. Catastrophic failure: fire, earthquake, etc.

- Data on disks lost and local backup lost!

=> Solution: Restore data from geographically distributed backup

Recovery in DBMS

- **Example (to illustrate consistency issues that can be introduced by failures)**

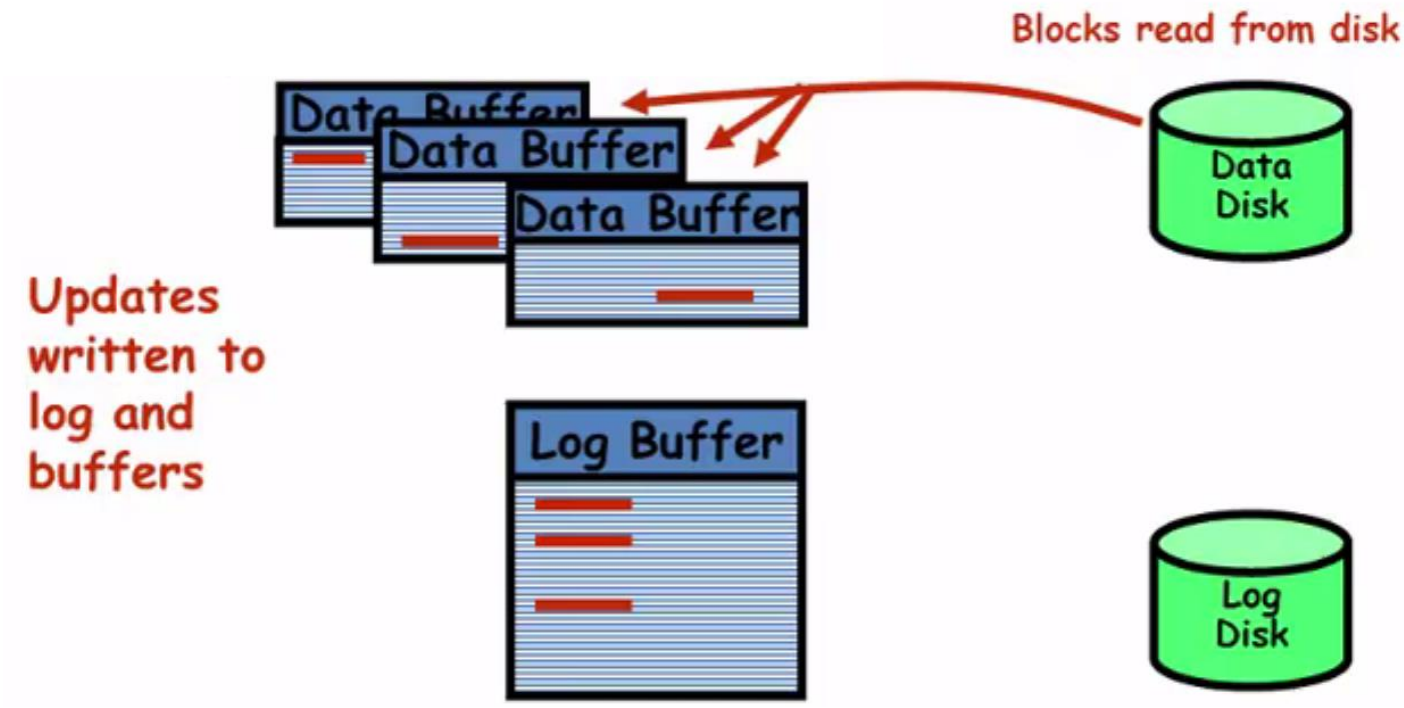
Balance transfer

```
decrement the balance of account X
    by $100;
increment the balance of account Y
    by $100;
```

- Scenario 1: Power failure after the first instruction
  - **Such failures may leave database in an inconsistent state with partial updates carried out**
  - Transfer of funds from one account to another should either complete or not happen at all

=> Database logs come to the rescue to allow undoing incomplete transaction!

Why do we need the log?



- Data items to be modified are first read into Data Buffers by the **Buffer Manager** (remember Buffer in volatile memory)
- Updates are happening "**in place**" i.e. data is overwritten on (deleted from) its data Buffer.
- Modified Data Buffers are **later** flushed (written) back to the disk (**Deferred Update**). The disk version of the data item is overwritten by the Buffer version (**In-place update**)

Challenge: REDO

Action	Buffer	Disk
Initially		0
T1 writes 1	1	0
T1 commits	1	0
CRASH		0

- Need to restore value 1 to item
  - Last value written by a committed transaction

Challenge: UNDO

Action	Buffer	Disk
Initially		0
T1 writes 1	1	0
Page flushed		1
CRASH		1

- Need to restore value 0 to item
  - Last value from a committed transaction

Transaction Log

- When deferred **in-place** update is used then log is necessary for recovery.
- Data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.
- These values are stored in a sequential file called Transaction log.
- Log File records all write operations in the order in which they occur
- Is an append-only file

Current database state = current state of data on disks + log

Log file entries

Types of records (entries) in log file:

- [start_transaction, T]: Records that transaction T has started execution.
- [T, X, old_value, new_value]: T has changed the value of item X from old_value to new_value.
- [commit, T]: T has completed successfully, and committed.
- [abort, T]: T has been aborted.

Log file entries (Cont.)

For **write_item** log entry, *old value* of item before modification (**BFIM** - Before Image) and the *new value* after modification (**AFIM** – After Image) are stored.

BFIM needed for UNDO, AFIM needed for REDO

A sample log

< START T1 >
<T1, A, 50, 25>
<T1, B, 250, 25>
<START T2>
<T1, A, 75, 50>
<T2, C, 35, 25>
<COMMIT T1>
<START T3>
<T3, E, 55, 25>
<T2, D, 45, 25>

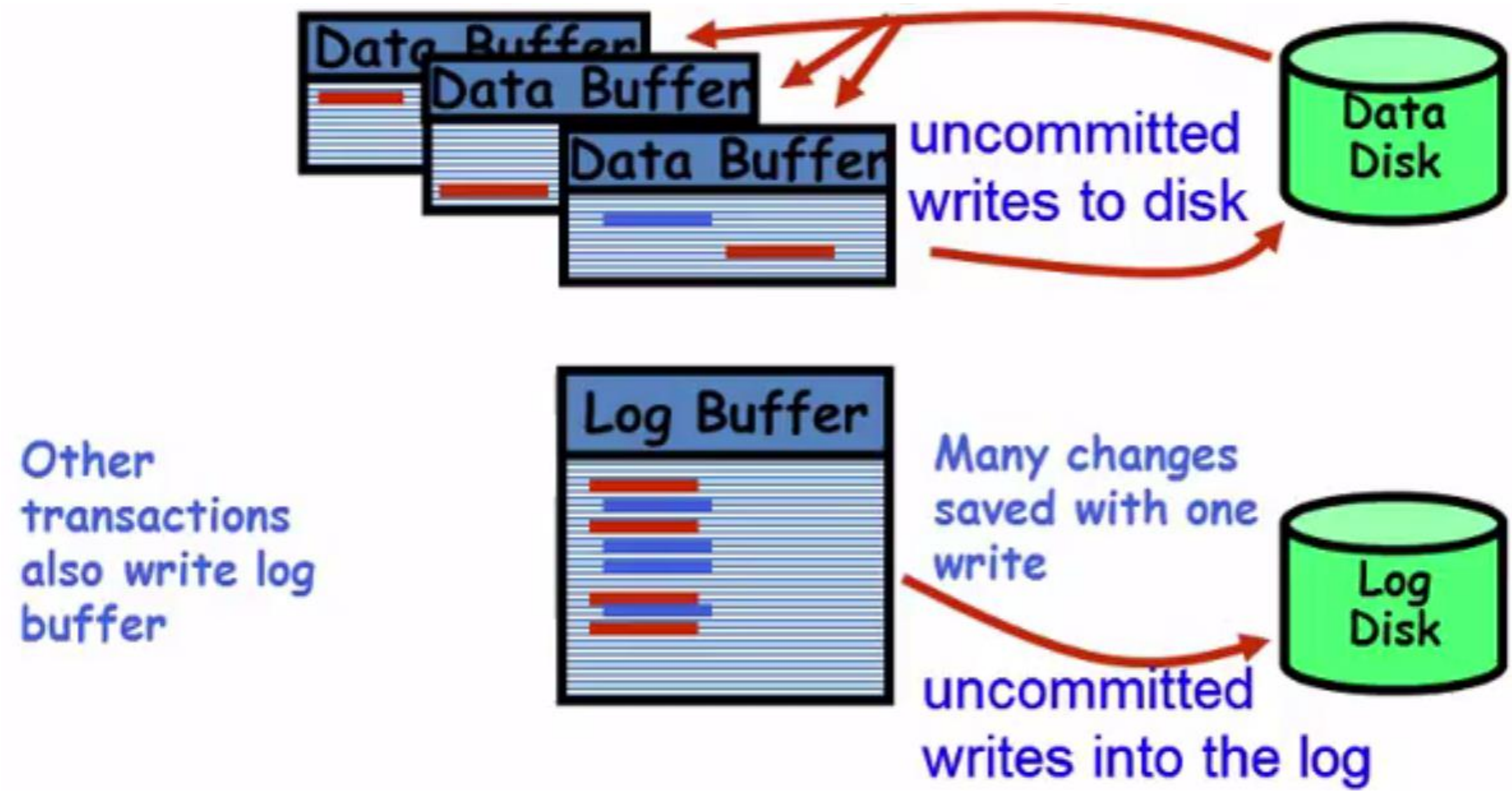
Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:

1. Must force the log record for an update *before* the corresponding data page gets to disk
2. Must write all log records for transaction *before commit returns*

- Property 1 guarantees Atomicity
- Property 2 guarantees Durability

Write-Ahead Logging (WAL)



Undo/Redo Logging

Update X → <T_i, X, old X value, New X value>

- Undo/Redo Logging Rules

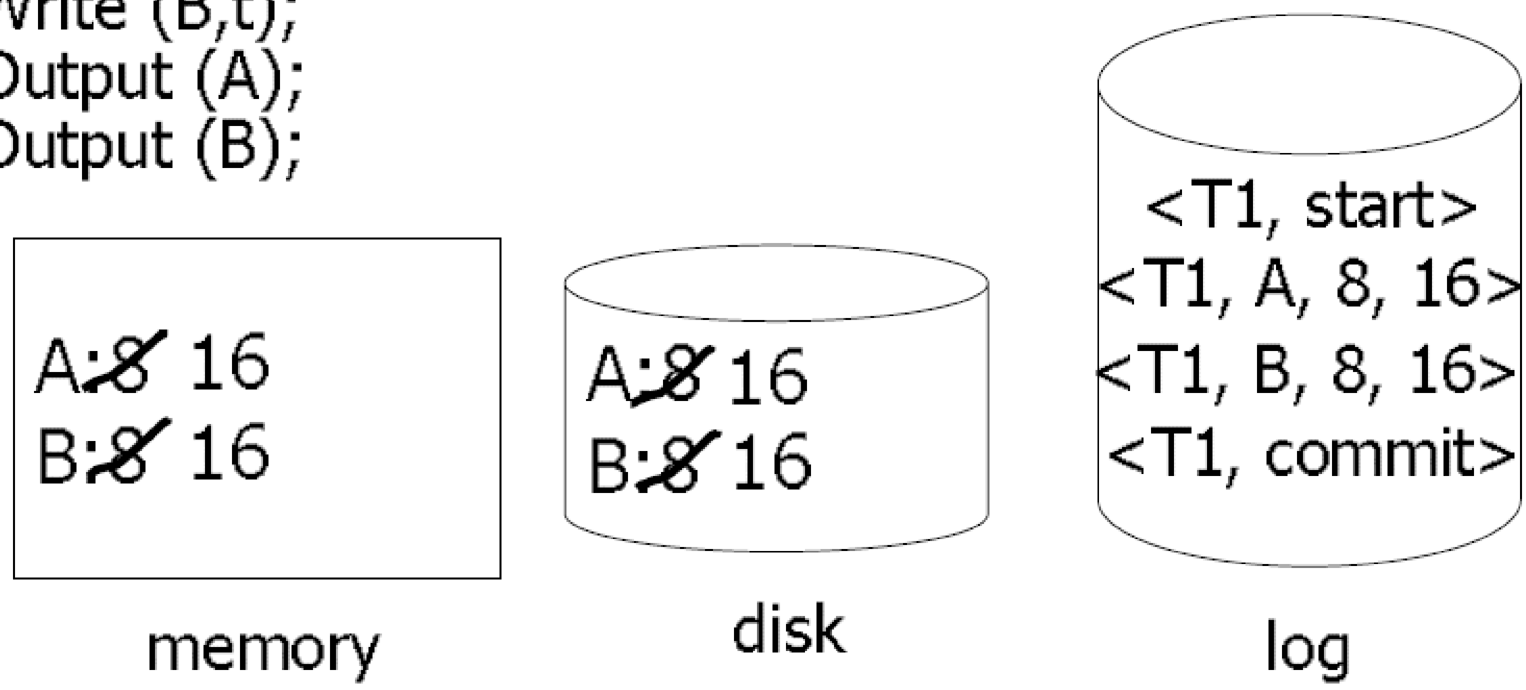
(1) Element X can be flushed before or after Ti commit

(2) Before modifying X on disk, all corresponding log records appear on disk

(3) Flush log before commit

Example

T1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



Recovery Process

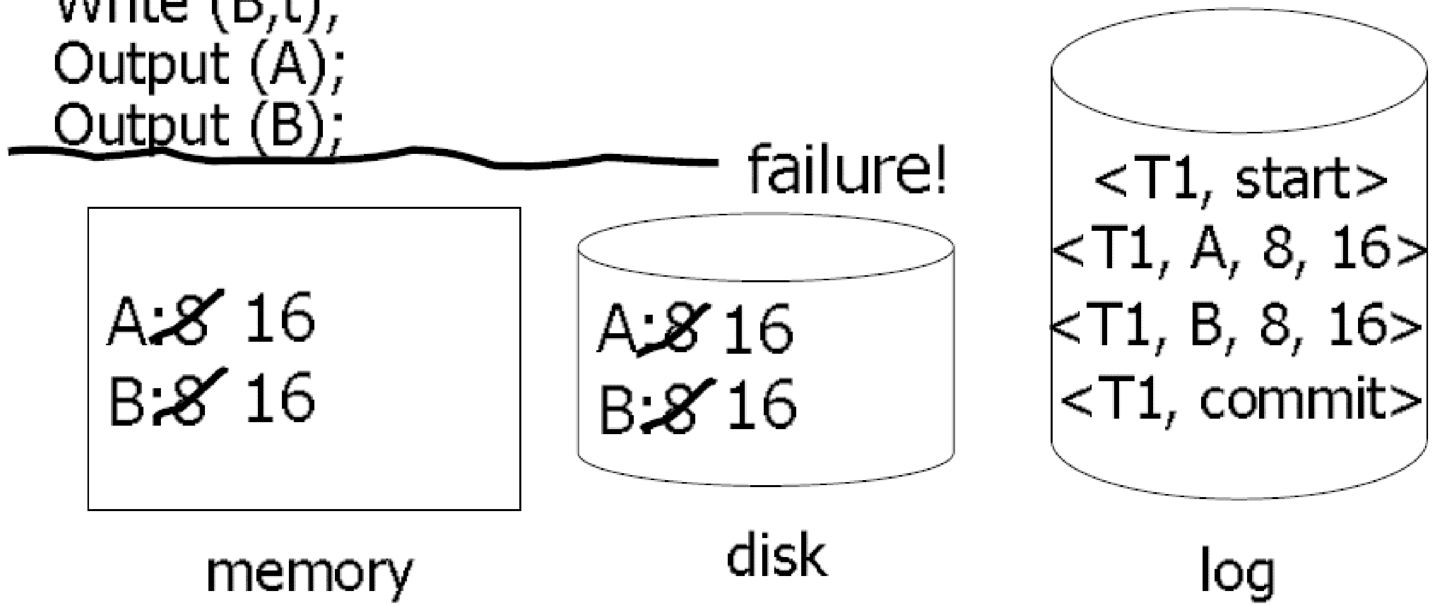
- The undo/redo recovery policy
  - Undo uncommitted transactions
  - Redo committed transactions
- **Backward pass** (end of log to the start)
  - Construct set **C** of transactions that committed
  - Undo all actions of transactions not in **C**
- **Forward pass** (start of log to the end)
  - Redo all actions of transactions in **C**

UNDO, REDO RECOVERY ACTIONS

- **Undo**: Restore all BFIMs from log to database on disk. UNDO proceeds backward in log
  - UNDO (roll-back) is needed for transactions that are not committed yet
- **Redo**: Restore all AFIMs from log to database on disk. REDO proceeds forward in log
  - REDO (roll-forward) is needed for committed transactions whose writes may have not yet been flushed from Buffer to Disk

Recovery Example 1

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

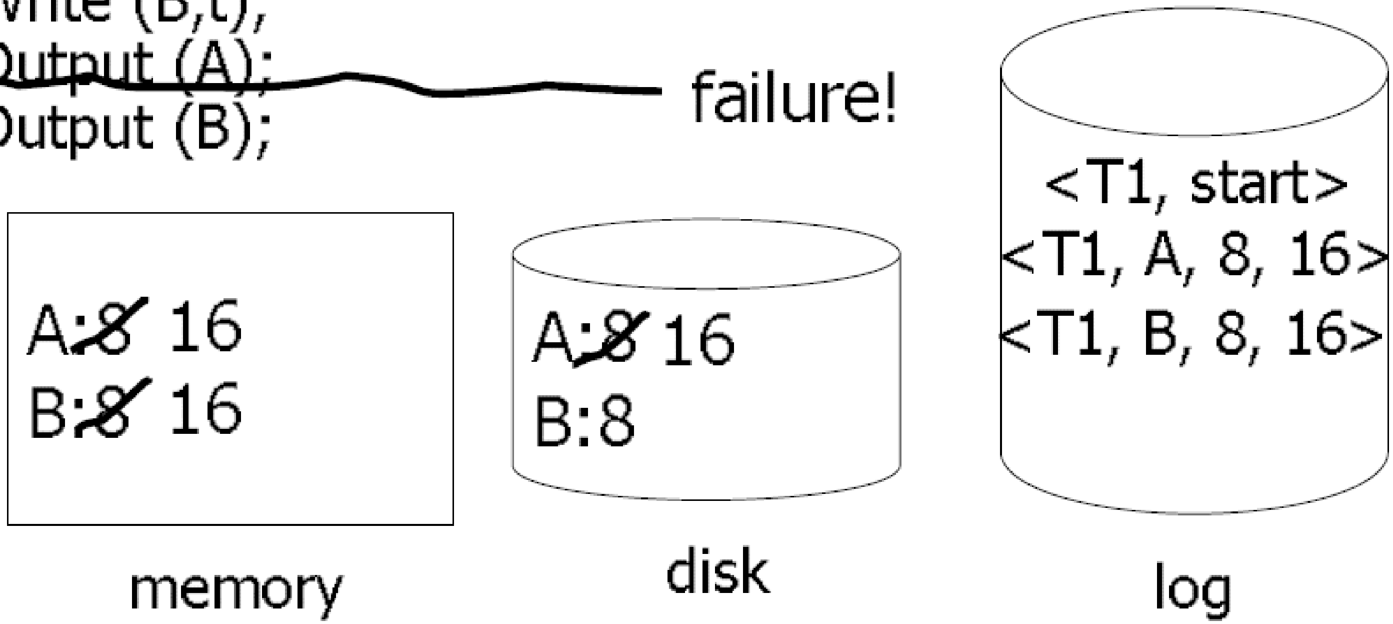


Recovery: T1 is committed. Redo: write 16 to both A and B on disk.

Recovery Example 2

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

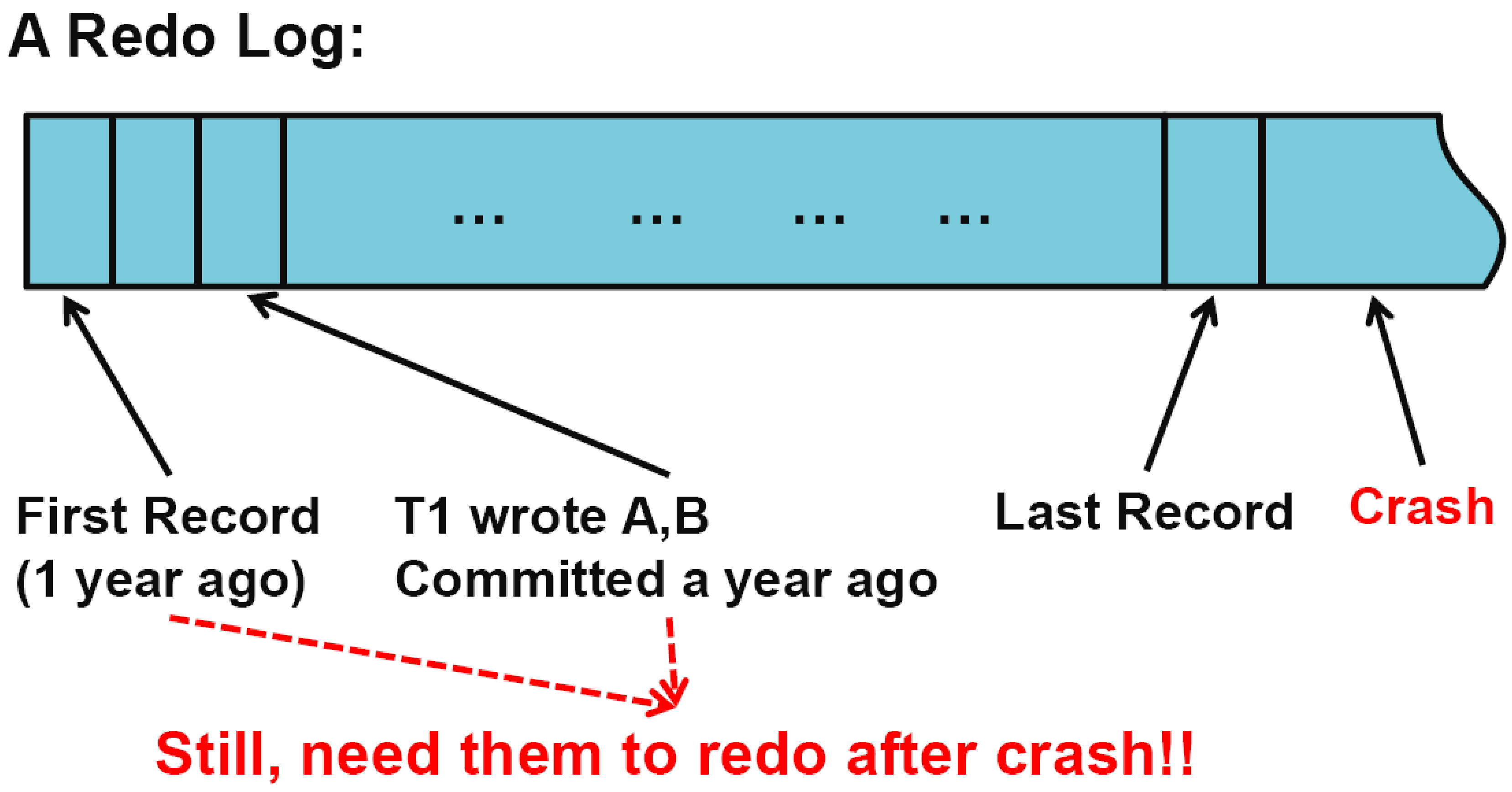
failure!



Recovery: T1 incomplete. Undo: old values ("8") of A and B are written to disk. Write <T1, abort> to the log.

Recovery is SLOW!

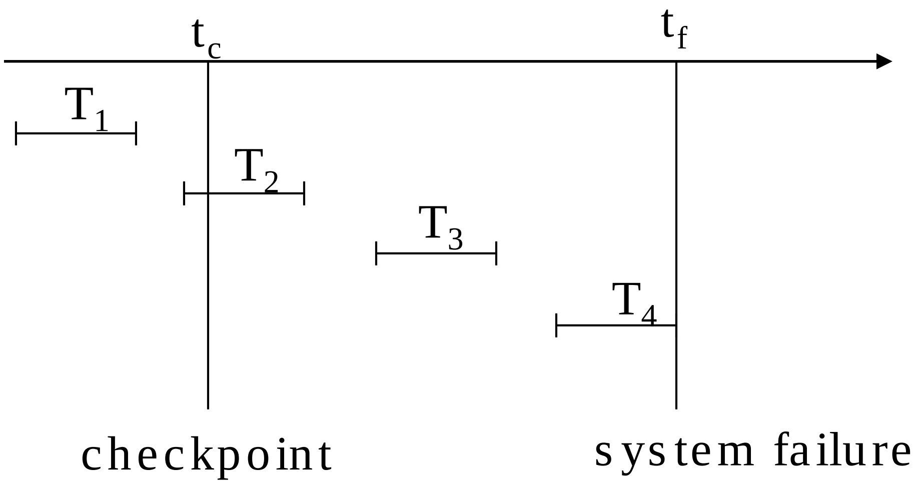
- Recovery can be slow if the log file is big. Hence Checkpointing is used to reduce recovery time



Checkpointing

- From time to time the database flushes its buffer to database disk to minimize the task of recovery.
- Following steps defines a checkpoint operation:
  1. Suspend execution of transactions temporarily
  2. Write "**start checkpoint**" listing all active transactions to log
  3. Force write dirty Data Buffers to disk whether or not their transactions have committed
  4. Write "end checkpoint" to log, save the log to disk.
  5. Resume normal transaction execution.
- During recovery redo or undo is required to transactions appearing after [checkpoint] record

Checkpointing Contd..



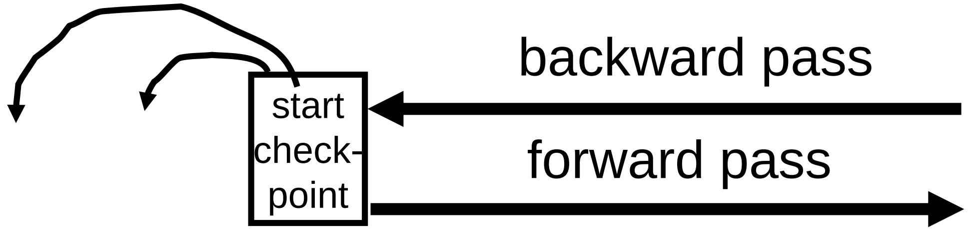
- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone
- T_4 undone

End Checkpoint

- If the system successfully performed the checkpointing (i.e., wrote **END CHECKPOINT** log entry to disk) then this means that all Dirty Data Buffers before the **START CHECKPOINT** must have been forced written to disk
  - If the crash happens <u>after</u> the **END CHECKPOINT** then we only redo actions of completed transactions <u>after</u> START CKPT (NOT from the start of the log file)
- This is basically the benefit of checkpointing i.e., avoid the need to redo ALL successful actions from the beginning of the log in order to reduce the recovery time

Recovery process

- Backwards pass (end of log to most recent checkpoint start)
  - construct set **S** of committed transactions
  - undo actions of transactions not in **S**
- Forward pass (latest checkpoint start to end of log)
  - redo actions of **S** transactions



Recovery Example 3

Redo/Undo Log:
<T1, START>
<T1, A, 4,5>
<T2, START>
<T1, COMMIT>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<T3, START>
<T3, D, 19, 20>
<END CKPT>
<T2, COMMIT>
<T3, COMMIT>

- T2 and T3 already committed.
- Since we see <END CKPT> first (backward), T1's changes must be on disk. So T1 can be ignored.
- Redo T2 and T3, forward.
  - Do NOT need to look at the log records before <START CKPT>
  - Reason: their changes are already on disk

Recovery Example 4

Redo/Undo Log:

<T1, START>

<T1, A, 4,5>

<T2, START>

<T1, COMMIT>

<T2, B, 9, 10>

<START CKPT(T2)>

<T2, C, 14, 15>

<T3, START>

<T3, D, 19, 20>

<END CKPT>

<T2, COMMIT>

- T2 committed.
- T3 incomplete
- Redo T2 (forward): C = 15 (on disk)
- Undo T3 (backward): D = 19 (on disk)

Recovery Example 5

Redo/Undo Log:
<T1, START>
<T1, A, 4,5>
<T2, START>
<T1, COMMIT>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<T3, START>
<T3, D, 19, 20>
<END CKPT>

- T2 and T3 incomplete
- Undo T3 (backward): D = 19 (on disk)
- Undo T2 (backward): C = 14 and B = 9 (on disk)
- Therefore, once we write the <END CKPT> log record, we still need to keep the earliest <Ti, START> log record for those Ti's that were active (when we started the CKPT) and incomplete during the ckpt.
  - In this case: <T2, START>

Recovery Example 6

Redo/Undo Log:

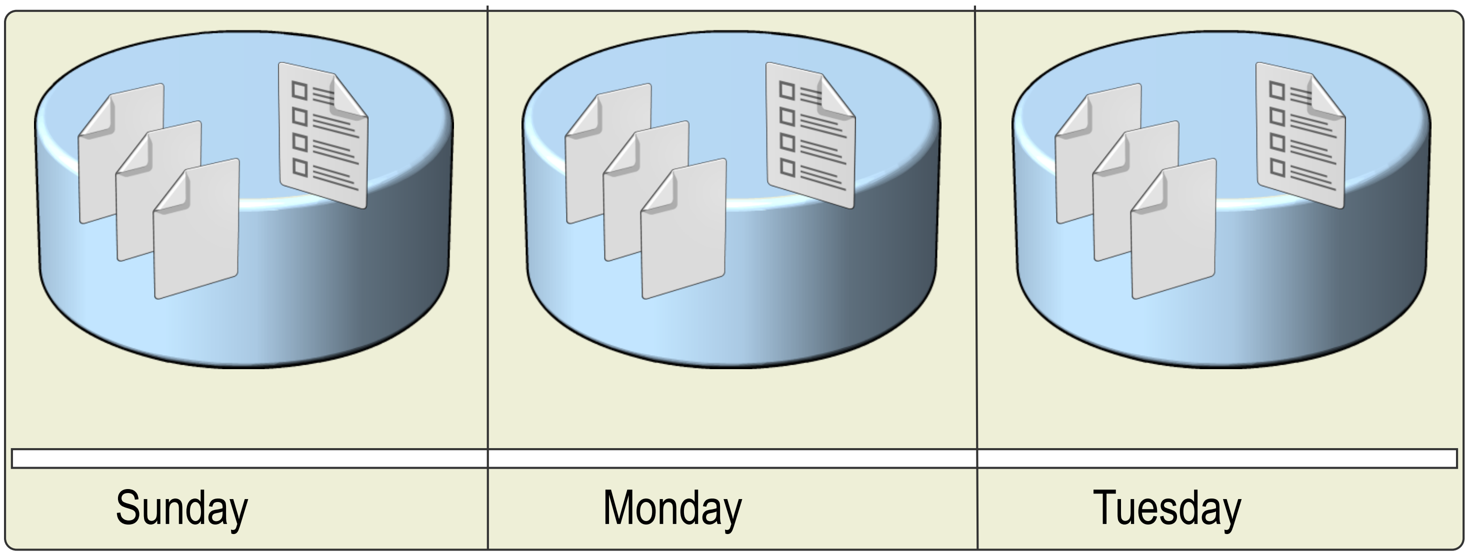
- <T1, START>
- <T1, A, 4, 5>
- <T2, START>
- <T1, COMMIT>
- <T2, B, 9, 10>
- **<START CKPT(T2)>**
- <T2, C, 14, 15>
- <T3, START>
- <T3, D, 19, 20>

- See <START CKPT(T2)> first (backward)
- T2 active and incomplete.
  - Undo (backward): C= 14 and B = 9 (on disk)
- T3 incomplete
  - Undo (backward): D = 19 (on disk)
- T1 complete
  - Redo (forward): A = 5 (on disk)

Recovery from Media Failure using Database Backup + Log Backup

- Restore full-database backup
- Restore differential backup takes after the restored backup
- Restore all logs from the most recent differential

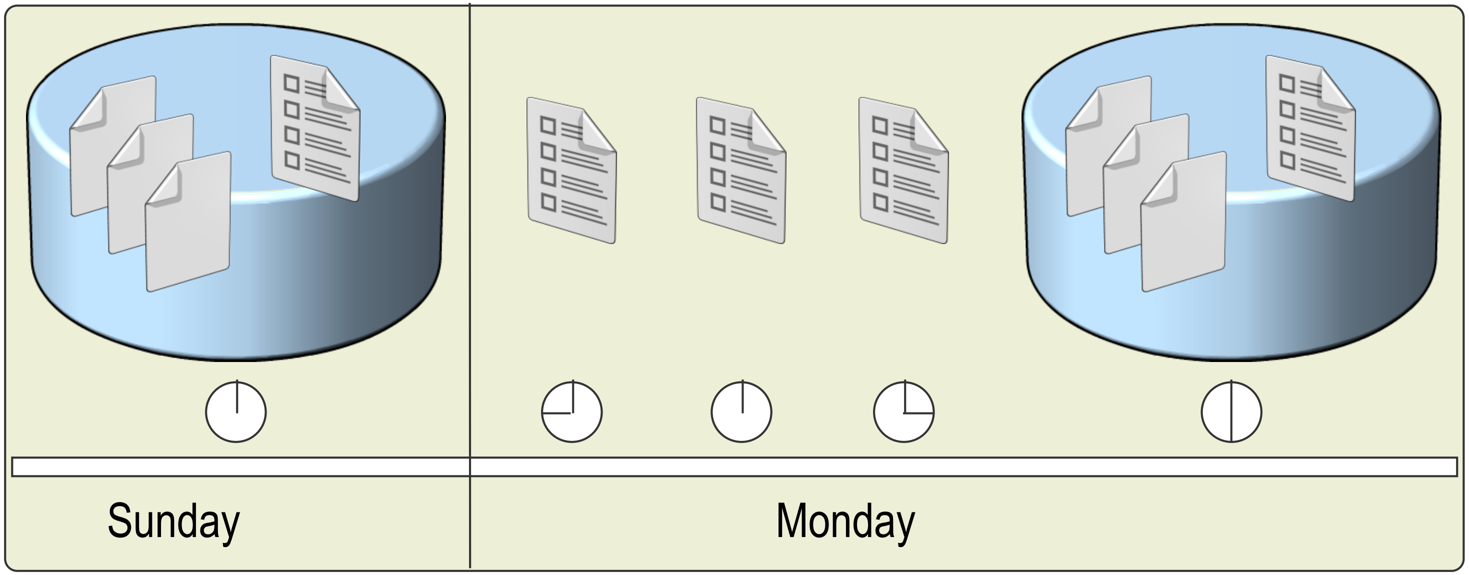
Full Database Backup



Full Database Backups:

- Backup all data and part of the log records
- Can be used to restore the whole database
- Permit recovery to backup times only

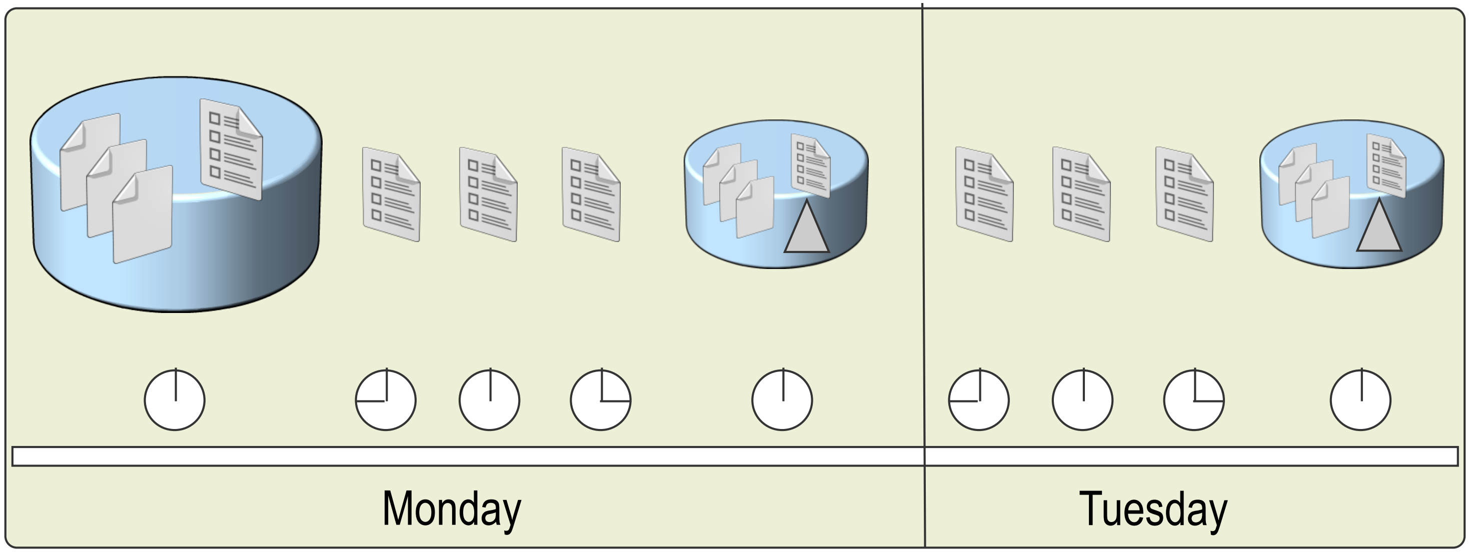
Transaction Log Backup



A Database and Transaction Log Backup Strategy:

- Involves at least full and transaction log backups
- Enables point in time recovery: you can recover the database to a specific point in time (for example, prior to entering unwanted data), or to the point of failure.
- Allows the database to be fully restored in the case of data file loss

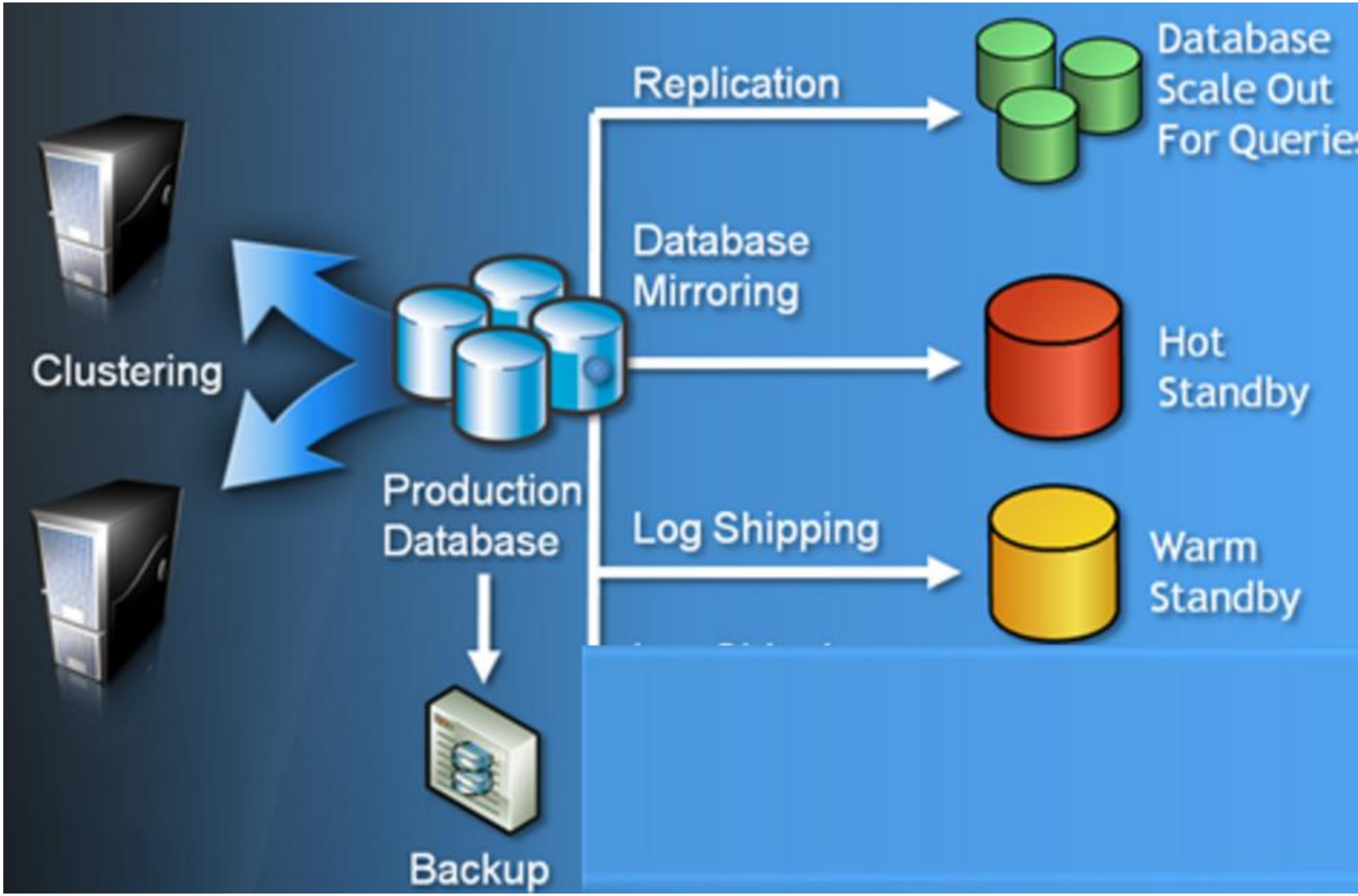
Differential Backup



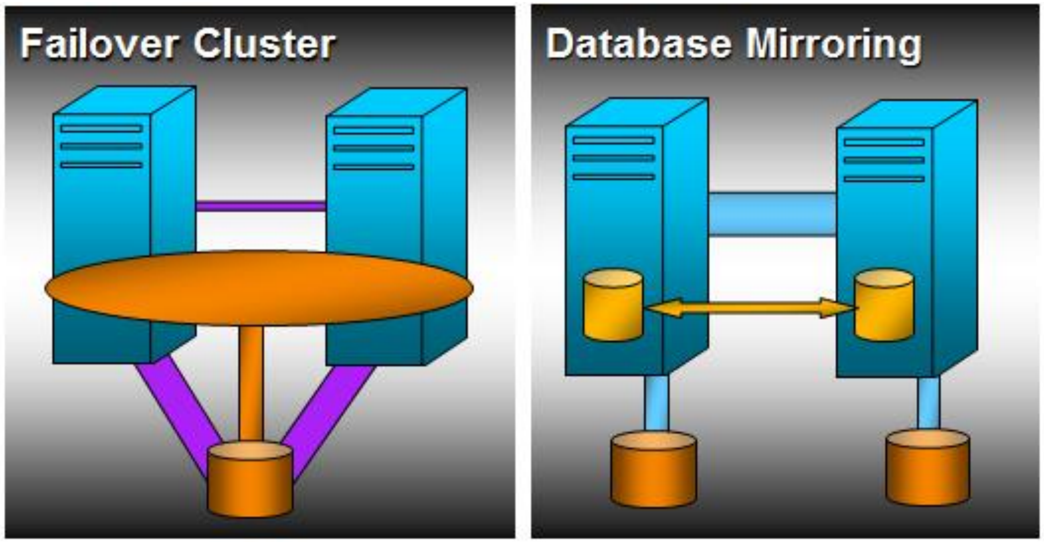
A Differential Backup:

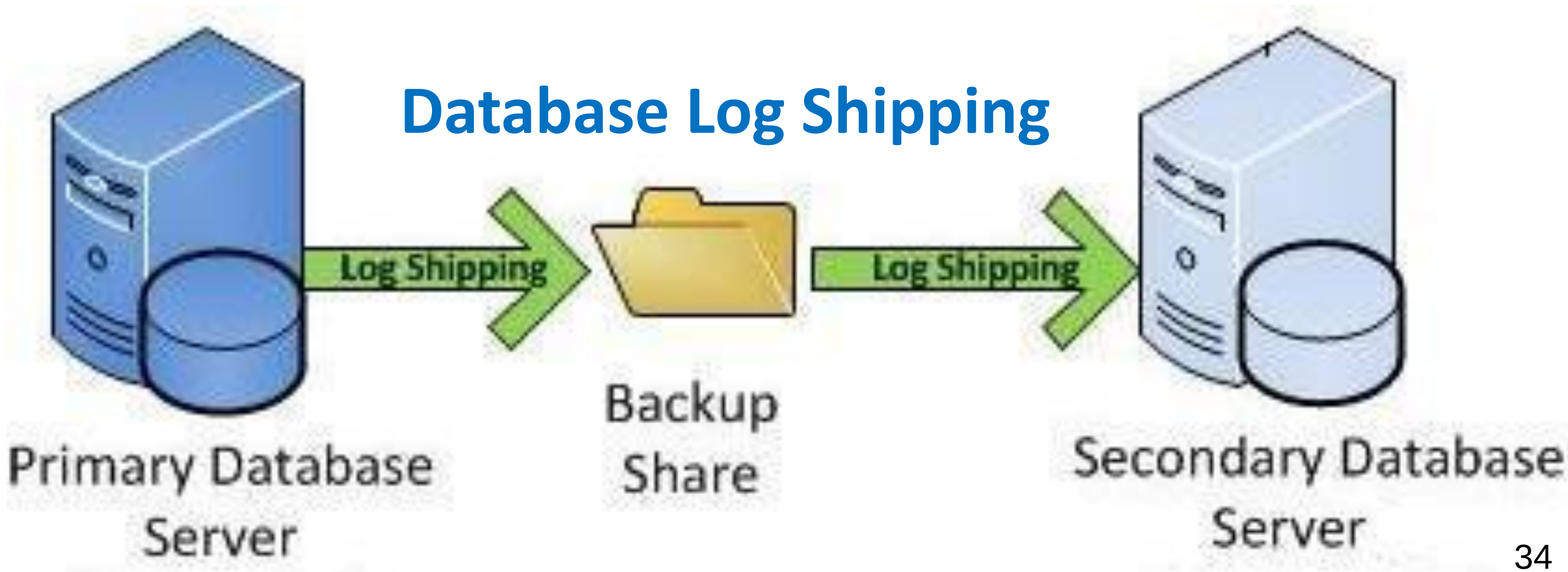
- A differential backup contains only the data that has changed since the differential base.
- Typically, differential backups are smaller and faster to create than the base of a full backup and also require less disk space to store backup images.

High Availability and Failover options

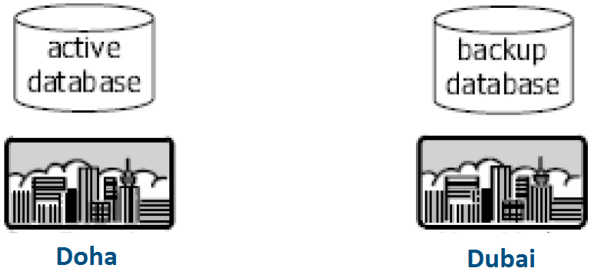


High Availability and Failover options





Catastrophic Failure



- Array of disks will not help in case of fire, earthquake, vandalism, viruses

=> Solution: Geographically distributed copies!

Summary

- **Recovery Manager** guarantees Atomicity & Durability.
- **Checkpointing** speeds-up recovery by limiting the amount of log to scan on recovery.
- Recovery from Media Failure using Database Backup + Log Backup.
- Other High Availability and Failover solutions could be used to ensure always-on availability