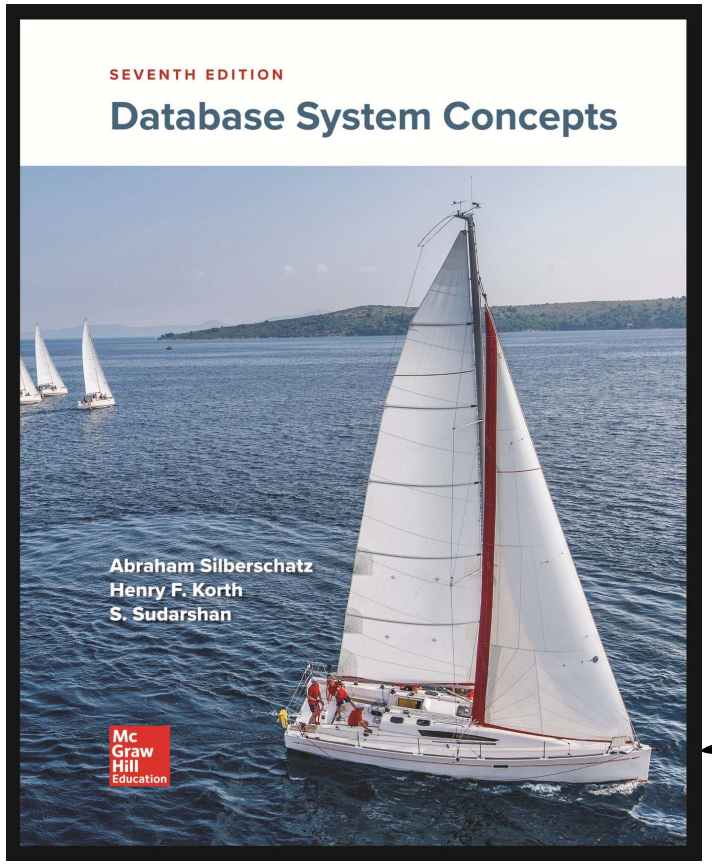
IF3140 – Sistem Basis Data
Recovery System

SEMESTER I TAHUN AJARAN 2024/2025

KNOWLEDGE & SOFTWARE ENGINEERING



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 19: Recovery System

KNOWLEDGE & SOFTWARE ENGINEERING

Objectives

Students are able to:

- Explain kinds of failures that can happen in a database system
- Explain stable storage and recovery task
- Explain when and why rollback is needed and how logging assures proper rollback
- Write a backup plan for a database

KNOWLEDGE & SOFTWARE ENGINEERING

Outline

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Remote Backup Systems

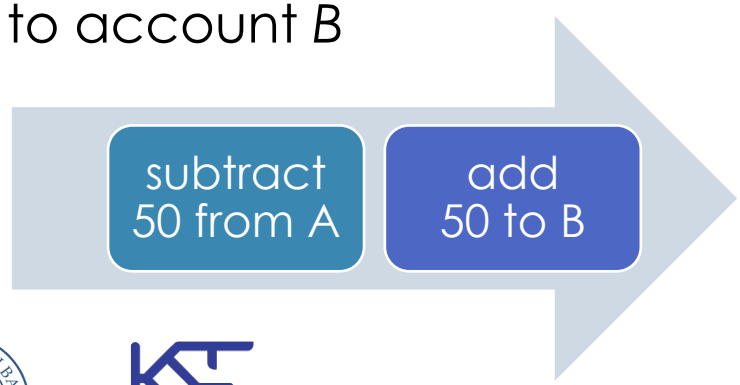
KNOWLEDGE & SOFTWARE ENGINEERING

Failure Classification

Transaction failure	• **Logical errors**: transaction cannot complete due to some internal error condition • **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
System crash	• **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	• Destruction is assumed to be detectable: disk drives use checksums to detect failures

Failure Example

Suppose transaction T_i transfers \$50 from account *A* to account *B*



Recovery Algorithm

Actions taken during normal transaction processing

to ensure enough information exists to recover from failures

Actions taken after a failure

to recover the database contents to a state that ensures atomicity, consistency and durability

KNOWLEDGE & SOFTWARE ENGINEERING

Storage Structure

Volatile storage:

- Does not survive system crashes

Nonvolatile storage:

- Survives system crashes
- 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

KNOWLEDGE & SOFTWARE ENGINEERING

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Block transfer can result in
  - Successful completion
  - Partial failure
  - Total failure
- Protecting storage media from failure during data transfer
  - Write the information onto the first physical block.
  - Then, write the same information onto the second physical block.
  - The output is completed only after the second write successfully completes.

Protecting Storage Media From Failure

Find inconsistent blocks:

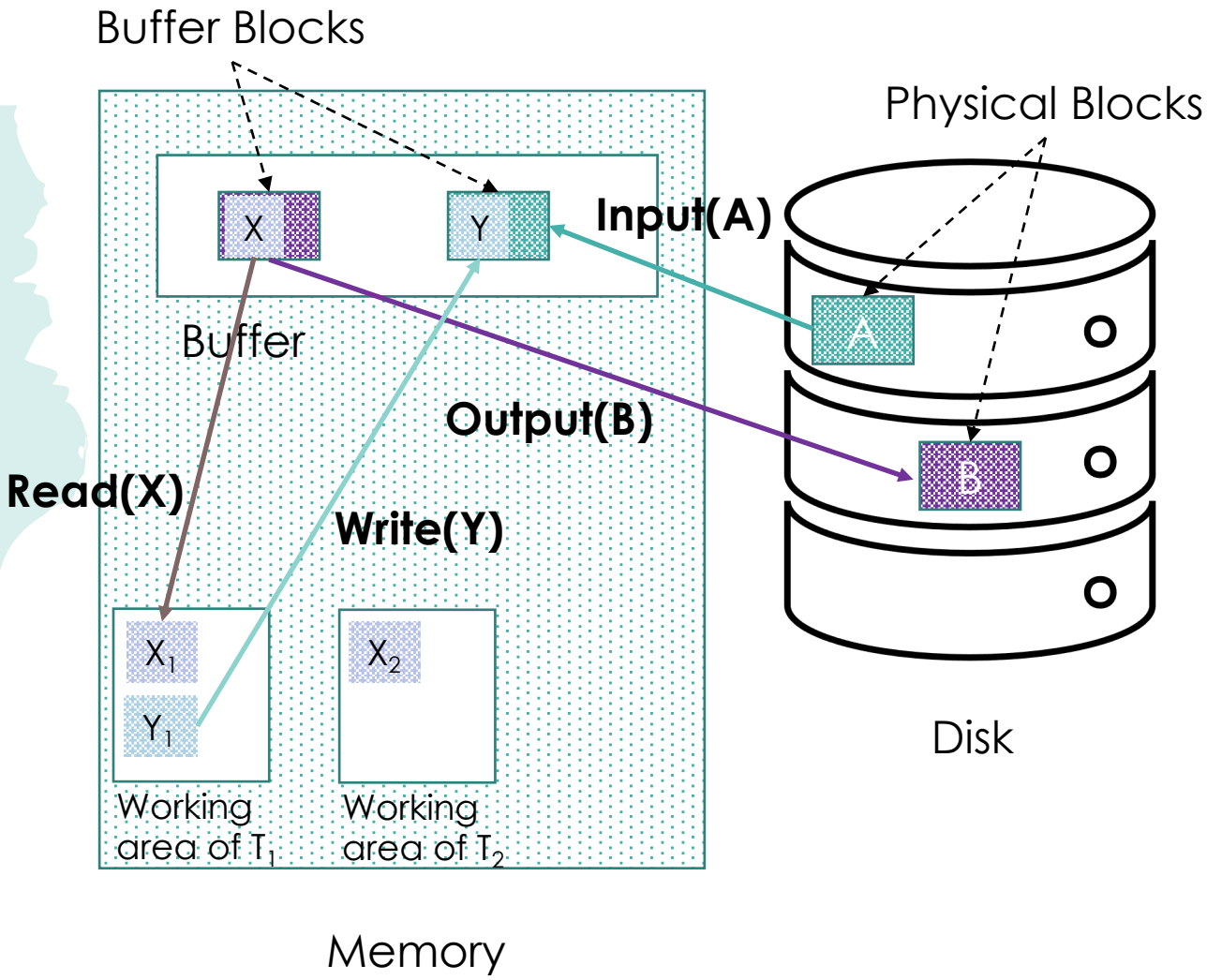
Expensive solution: Compare the two copies of every disk block	*Better solution*: Record in-progress disk writes on non-volatile storage

Recovery:

If either copy is detected to have an error, overwrite it	If both have no error but are different, overwrite the second block

KNOWLEDGE & SOFTWARE ENGINEERING

Data Access

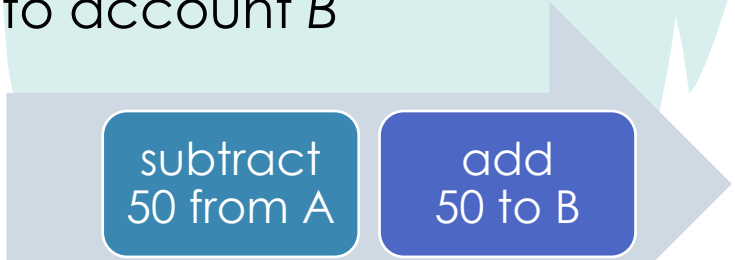


Recovery and Atomicity

LOG-BASED RECOVERY MECHANISMS IN DETAIL

LESS USED ALTERNATIVE: **SHADOW-COPY** AND **SHADOW-PAGING**

Suppose transaction T_i transfers \$50 from account *A* to account *B*



KNOWLEDGE & SOFTWARE ENGINEERING

Log-Based Recovery

A **log**	• A sequence of **log records** • The **log** is kept on stable storage
When transaction T_i starts	• <T_i **start**> log record
Before T_i executes **write**(X)	• <T_i, X, V_1, V_2> log record
When T_i finishes	• <T_i **commit**> log record

Transaction T_{10} transfers \$50 from account *A* to account *B*

Value of A =1000 and B = 2000

<T_{10} start>

< T_{10}, A, 1000, 950>

< T_{10}, B, 2000, 2050>

< T_{10} commit>

- Immediate database modification
- Deferred database modification

KNOWLEDGE & SOFTWARE ENGINEERING

Immediate Database Modification

Update log record must be written *before* database item is written

Output of updated blocks:

- can take place at any time
- order can be different from the order in which they are written

Transaction Commit:

- A transaction is said to have committed when its commit log record is output to stable storage
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate Database Modification Example

Log	Write	Output
$<T_0$ **start**$>$		
$<T_0$, A, 1000, 950$>$		
$<T_0$, B, 2000, 2050$>$		
	$A = 950$ $B = 2050$	
$<T_0$ **commit**$>$		
$<T_1$ **start**$>$		
$<T_1$, C, 700, 600$>$		
	$C = 600$	
		B_B , B_C
$<T_1$ **commit**$>$		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

KNOWLEDGE & SOFTWARE ENGINEERING

Concurrency Control and Recovery

All concurrent transactions share a single disk buffer and a single log

- A buffer block can have data items updated by one or more transactions

We assume strict two-phase commit

- i.e., the updates of uncommitted transactions should not be visible to other transactions

Log records of different transactions may be interspersed in the log

KNOWLEDGE & SOFTWARE ENGINEERING

Recovering from Failure

Transaction T_i needs to be **undone**

- Contains the record <T_i **start**>,
- But does not contain either the record <T_i **commit**> *or* <T_i **abort**>.

Transaction T_i needs to be **redone**

- Contains the records <T_i **start**>
- And contains the record <T_i **commit**> *or* <T_i **abort**>

KNOWLEDGE & SOFTWARE ENGINEERING

Recovering from Failure

Transaction T_i needs to be undone

- Contains the record <T_i **start**>,
- But does not contain either the record <T_i **commit**> *or* <T_i **abort**>.

Transaction T_i needs to be redone

- Contains the records <T_i **start**>
- And contains the record <T_i **commit**> *or* <T_i **abort**>

undo(T_i) -- restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i

- Each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out.
- When undo of a transaction is complete, a log record <T_i **abort**> is written out.

redo(T_i) -- sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

- No logging is done in this case

KNOWLEDGE & SOFTWARE ENGINEERING

Immediate DB Modification Recovery Example

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>

Recovery action:
undo T_0
- Log <T_0, B, 2000>
- Restore B to 2000
- Log <T_0, A, 1000>
- Restore A to 1000
- Log <T_0 abort>

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

Recovery action:
redo T_0 and undo T_1
- Set A to 950
- Set B to 2050
- Log <T_1, C, 700>
- Restore C to 700
- Log <T_1 abort>

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

Recovery action:
redo T_0 and redo T_1
- Set A to 950
- Set B to 2050
- Set C to 600

KNOWLEDGE & SOFTWARE ENGINEERING

Checkpoints

Streamline recovery procedure by periodically performing **checkpointing**

Output all log records currently residing in main memory onto stable storage.

Output all modified buffer blocks to the disk.

Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.

All updates are stopped while doing checkpointing

KNOWLEDGE & SOFTWARE ENGINEERING

Checkpoints – On Recovery

During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

- Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
- Only transactions that are in *L* or started after the checkpoint need to be redone or undone
- Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

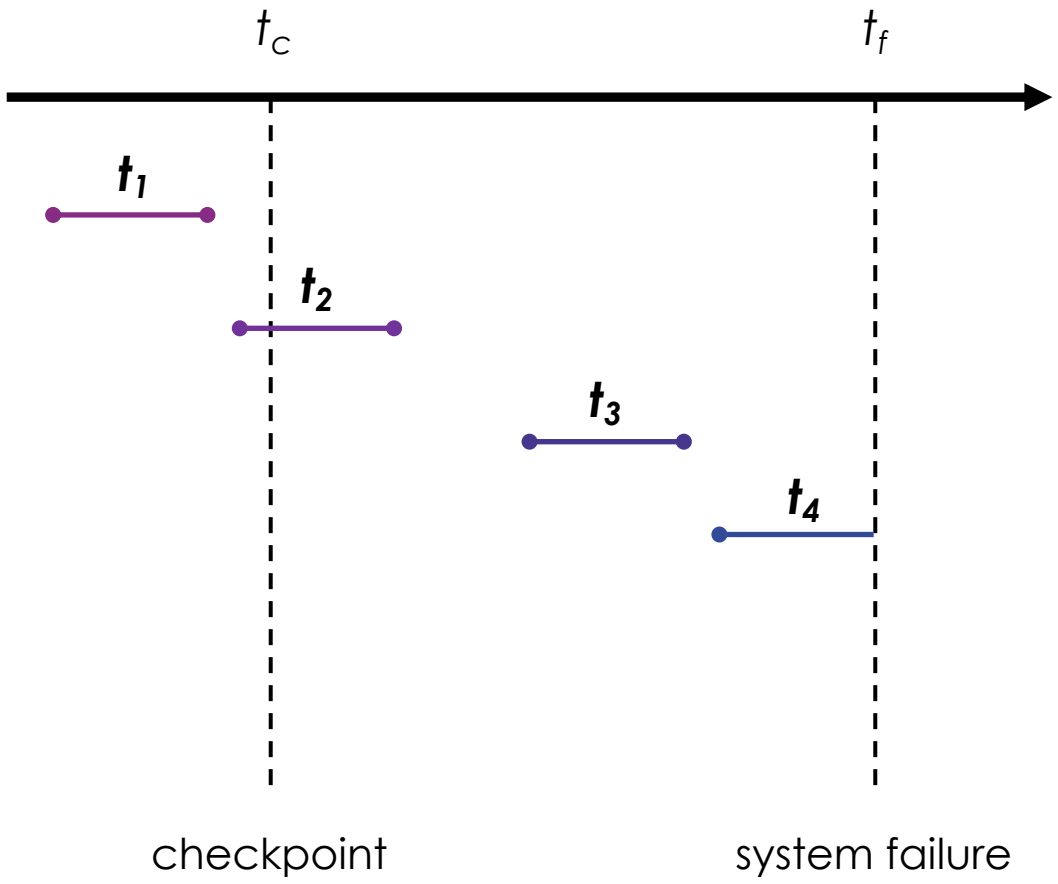
Some earlier part of the log may be needed for undo operations

- Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
- Parts of log prior to earliest <T_i **start**> record above are not needed for recovery and can be erased whenever desired.

KNOWLEDGE & SOFTWARE ENGINEERING

Example of Checkpoints

T_1 can be ignored
T_2 and T_3 redone.
T_4 undone



KNOWLEDGE & SOFTWARE ENGINEERING

Recovery Algorithm

So far: we covered key concepts

Now: we present the components of the basic recovery algorithm

Later: we present extensions to allow more concurrency

KNOWLEDGE & SOFTWARE ENGINEERING

Recovery Algorithm

Logging
(during normal operation)

- <T_i **start**> at transaction start
- <T_i, X_j, V_1, V_2> for each update
- <T_i **commit**> at transaction end

Transaction rollback
(during normal operation)

- Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
  - Perform the undo by writing V_1 to X_j,
  - Write a log record <T_i , X_j, V_1>
    - such log records are called **compensation log records**
- Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

KNOWLEDGE & SOFTWARE ENGINEERING

Recovery Algorithm – Recovery from Failure

Redo phase

- replay updates of **all** transactions, whether they committed, aborted, or are incomplete

Undo phase

- undo all incomplete transactions

KNOWLEDGE & SOFTWARE ENGINEERING

Recovery Algorithm – Recovery from Failure

Redo phase

Find last <**checkpoint** *L*> record, and set undo-list to *L*.

Scan forward from above <**checkpoint** *L*> record

- Whenever a record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ is found, redo it by writing V_2 to X_j
- Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
- Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Undo phase

KNOWLEDGE & SOFTWARE ENGINEERING

Recovery Algorithm – Recovery from Failure

Redo phase

Undo phase

Scan log backwards from end

Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list → transaction rollback

- perform undo by writing V_1 to X_j.
- write a log record $<T_i, X_j, V_1>$

Whenever a log record $<T_i$ **start**$>$ is found where T_i is in undo-list,

- Write a log record $<T_i$ **abort**$>$
- Remove T_i from undo-list

Stop when undo-list is empty, i.e., $<T_i$ **start**$>$ has been found for every transaction in undo-list

After undo phase completes, normal transaction processing can commence

KNOWLEDGE & SOFTWARE ENGINEERING

Example of Recovery