

# Database System

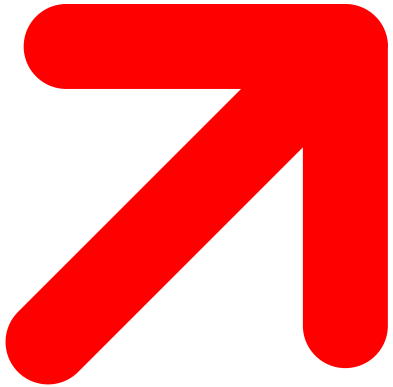
## 12 | Backup & Recovery

Tahun Ajar Ganjil 2024/2025

Oleh:  
Tim Dosen

---

# Goals of the Meeting

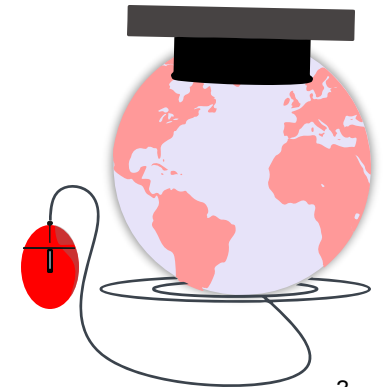


Students are able to explain backup and recovery



# OUTLINES

- Backup
- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery



# BACKUP



# BACKUP MODES

- **Hot backup**

- allows backup of the database while the database is running and available to users.
- performance degrades during the backup period.
- takes longer than a cold backup

- **Cold backup**

- requires database shutdown before backup begins
- physical files are backed up while shutdown
- database is unavailable to users during backup period
- faster than a hot backup



# ORACLE BACKUP OPTIONS

- Off-line physical backup
  - is a cold backup technique
  - copies data files, log files, init files, and control files after shutdown
- On-line physical backup
  - is a hot backup technique
  - temporarily sets tablespaces into backup mode
- Logical Backup
  - is a hot backup technique
  - exports all or part of the database by creating SQL scripts necessary to
  - recreate the objects



# OFF-LINE PHYSICAL BACKUP

- Shut down the database
- Backup the data files
- Backup the control files
- Backup the online redo log files or the archive log files
- Backup the initialization and password files
- Restart the database



# ON-LINE PHYSICAL BACKUP

- Requires the DB to be in Archive Log Mode
- For each tablespace
  - Set the tablespace into a backup state
  - Backup the datafiles for that tablespace
  - Restore the tablespace to its normal state
- Backup the archived redo log files
  - Stop the archiving process
  - Note which files are in the arch log directory
  - Restart the archiving process
  - Backup the archived redo log files
  - Delete the archived redo log files
- Backup the control file using the “Alter database backup controlfile command”
- Backup the initialization and password files





# LOGICAL BACKUP

- Essentially creates SQL scripts that will recreate database objects
- Use Oracle Export Utility to create .dmp file
- Use Oracle Import Utility to selectively import database objects from
  - the .dmp file
- Can perform
  - Complete database backup
  - User (schema) backup
  - Individual table backup
- Used in conjunction with a recovery server to recover selected objects (not entire database)

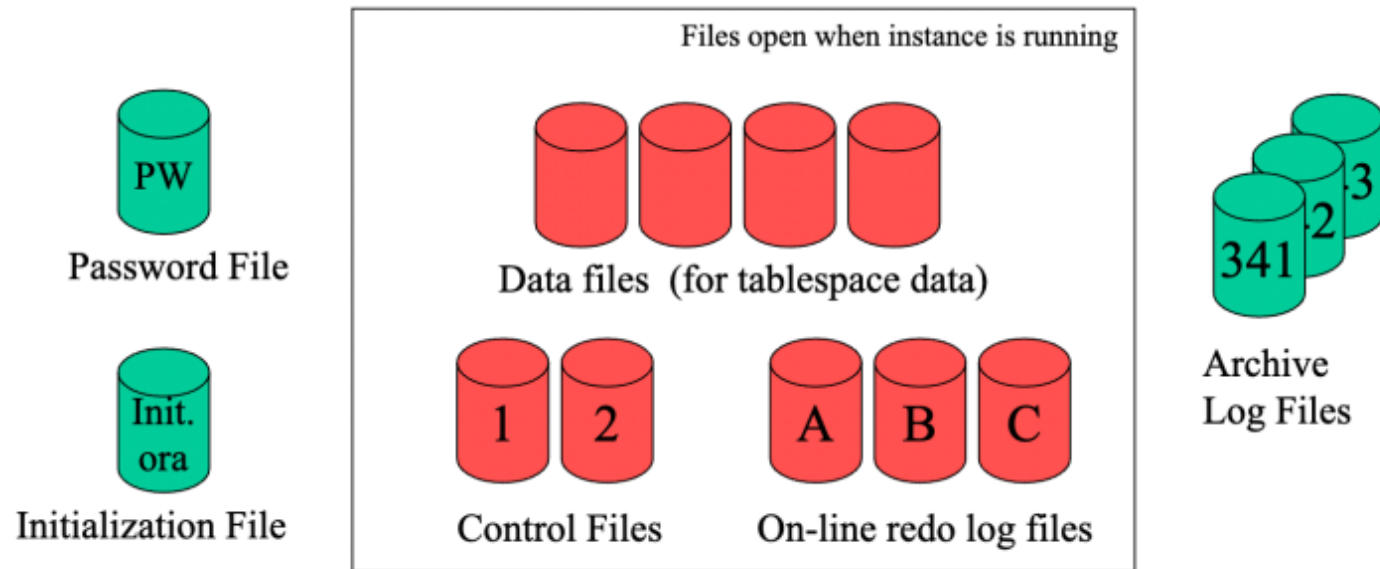


# BACKUP TYPES

- Complete (Full)
  - copy all database and related files
  - delete the archive log files
- Cumulative (Differential)
  - copy blocks that have changed since last full backup or
  - copy all archive log files generated since last full backup
- Incremental
  - copy blocks that have change since the last partial backup or
  - copy all log files generated since last partial backup
- Complete (Copy)
  - copy all target data
  - Don't include the set in backup set logic



# ORACLE FILES OVERVIEW



# FAILURE CLASSIFICATION



# INTRODUCTION

- A computer system is subject to failure from a variety of causes.
- In any failure, **information may be lost**.
- The database system must take actions in advance to ensure that the **atomicity** and **durability** properties of transactions are preserved.
- 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.
- The recovery scheme must also support high availability: the database should be usable for a very high percentage of time.



# 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:** a power failure or other hardware or software failure causes the system to 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:** a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures



# STORAGE STRUCTURE



# 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





# 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.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.



# PROTECTING STORAGE MEDIA FROM FAILURE (CONT.)

Copies of a block may differ due to failure during output operation.

To recover from failure:

1. First find inconsistent blocks:

1. *Expensive solution:* Compare the two copies of every disk block.

2. *Better solution:*

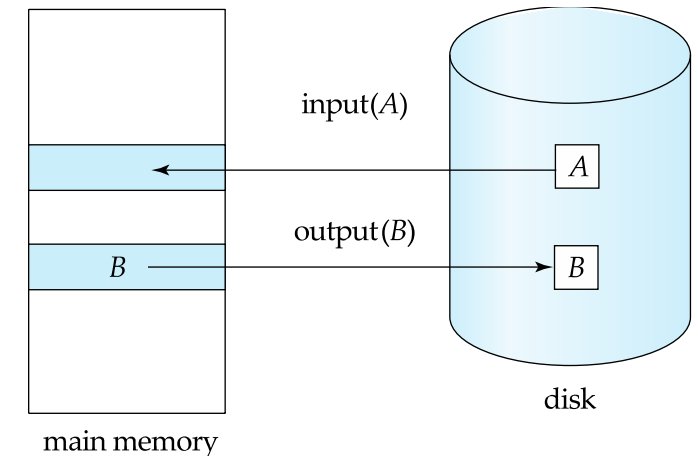
- Record in-progress disk writes on non-volatile storage (Flash, Non-volatile RAM or special area of disk).
- Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
- Used in hardware RAID systems

2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.



# DATA ACCESS

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input** ( $B$ ) transfers the physical block  $B$  to main memory.
  - **output** ( $B$ ) transfers the buffer block  $B$  to the disk and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

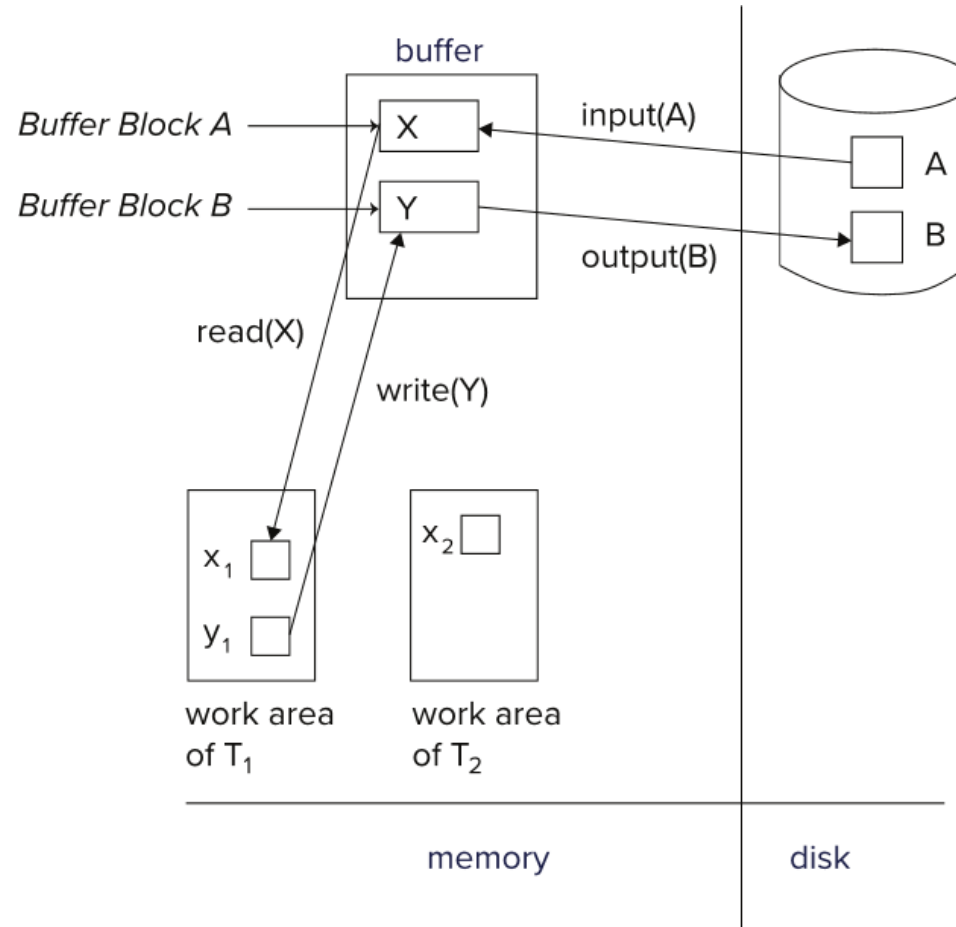


## DATA ACCESS (CONT.)

- Each transaction  $T_i$  has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - $T_i$ 's local copy of a data item  $X$  is called  $x_i$ .
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**( $X$ ) assigns the value of data item  $X$  to the local variable  $x_i$ .
  - **write**( $X$ ) assigns the value of local variable  $x_i$  to data item  $\{X\}$  in the buffer block.
  - Note: **output**( $B_x$ ) need not immediately follow **write**( $X$ ). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**( $X$ ) before accessing  $X$  for the first time (subsequent reads can be from local copy)
  - **write**( $X$ ) can be executed at any time before the transaction commits



# EXAMPLE OF DATA ACCESS



# RECOVERY AND ATOMICITY



# RECOVERY ALGORITHMS

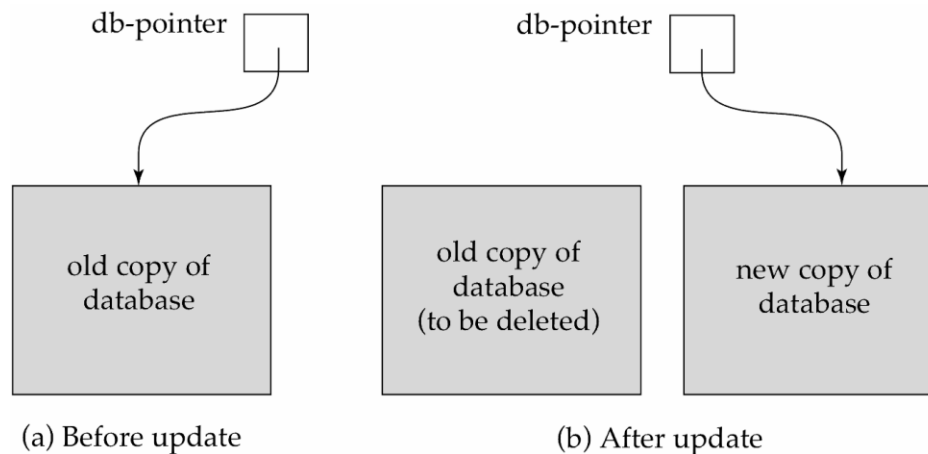
- Suppose transaction  $T_i$  transfers \$50 from account  $A$  to account  $B$ 
  - Two updates: subtract 50 from  $A$  and add 50 to  $B$
- Transaction  $T_i$  requires updates to  $A$  and  $B$  to be output to the database.
  - A failure may occur **after one of these modifications have been made** but before both of them are made.
  - Modifying the database **without** ensuring that the transaction will **commit** may leave the database in an inconsistent state
  - **Not modifying** the database may result in **lost updates if failure** occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions **taken during normal** transaction processing to ensure enough information exists to recover from failures
  2. Actions **taken after a failure to recover** the database contents to a state that ensures atomicity, consistency and durability



# RECOVERY AND ATOMICITY

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
- Less used alternative: **shadow-copy** and **shadow-paging** (brief details in book)

## shadow-copy





# LOG-BASED RECOVERY



# LOG - BASED RECOVERY

- A **log** is a sequence of **log records**. The records keep information about update activities on the database.
  - The **log** is kept on stable storage
- When transaction  $T_i$  starts, it registers itself by writing a  
 $\langle T_i \text{ start} \rangle$  log record
- Before  $T_i$  executes **write**( $X$ ), a log record  
 $\langle T_i, X, V_1, V_2 \rangle$   
is written, where  $V_1$  is the value of  $X$  before the write (the **old value**), and  $V_2$  is the value to be written to  $X$  (the **new value**).
- When  $T_i$  finishes its last statement, the log record  $\langle T_i \text{ commit} \rangle$  is written.
- Two approaches using logs
  - **Immediate** database modification
  - **Deferred** database modification.



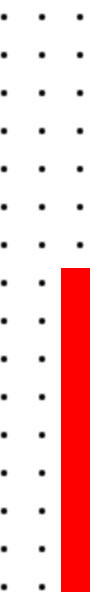
# IMMEDIATE DATABASE MODIFICATION

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to disk can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.



# DEFERRED DATABASE MODIFICATION

- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy



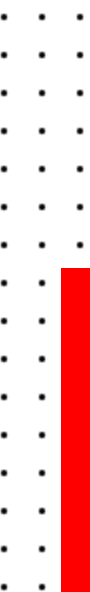
# TRANSACTION COMMIT

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- 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	Database
$\langle T_0 \text{ start} \rangle$	
$\langle T_0, A, 1000, 950 \rangle$	
$\langle T_0, B, 2000, 2050 \rangle$	
	$A = 950$
	$B = 2050$
$\langle T_0 \text{ commit} \rangle$	
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 700, 600 \rangle$	
	$C = 600$
$\langle T_1 \text{ commit} \rangle$	



# CONCURRENCY CONTROL AND RECOVERY

- With concurrent transactions, all 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 that *if a transaction  $T_i$  has modified an item, no other transaction can modify the same item until  $T_i$  has committed or aborted*
  - i.e., the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise, how to perform undo if  $T_1$  updates A, then  $T_2$  updates A and commits, and finally  $T_1$  has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.



# UNDO AND REDO OPERATIONS

- **Undo and Redo of Transactions**

- **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  $\langle T_i, X, V \rangle$  is written out
  - When undo of a transaction is complete, a log record  $\langle T_i, \text{abort} \rangle$  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





# RECOVERING FROM FAILURE

- When recovering after failure:
  - Transaction  $T_i$  needs to be undone if the log
    - Contains the record  $\langle T_i \text{ start} \rangle$ ,
    - But does not contain either the record  $\langle T_i \text{ commit} \rangle$  or  $\langle T_i \text{ abort} \rangle$ .
  - Transaction  $T_i$  needs to be redone if the log
    - Contains the records  $\langle T_i \text{ start} \rangle$
    - And contains the record  $\langle T_i \text{ commit} \rangle$  or  $\langle T_i \text{ abort} \rangle$



## RECOVERING FROM FAILURE (CONT.)

- Suppose that transaction  $T_i$  was undone earlier and the  $\langle T_i, \mathbf{abort} \rangle$  record was written to the log, and then a failure occurs,
- On recovery from failure transaction  $T_i$  is redone
  - Such a **redo** redoes all the original actions of transaction  $T_i$  *including the steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly



# IMMEDIATE DB MODIFICATION RECOVERY

## EXAMPLE

Below we show the log as it appears at three instances of time.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 700, 600 \rangle$	$\langle T_1, C, 700, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

Recovery actions in each case above are:

- (a) undo ( $T_0$ ): B is restored to 2000 and A to 1000, and log records  $\langle T_0, B, 2000 \rangle$ ,  $\langle T_0, A, 1000 \rangle$ ,  $\langle T_0, \mathbf{abort} \rangle$  are written out
- (b) redo ( $T_0$ ) and undo ( $T_1$ ): A and B are set to 950 and 2050 and C is restored to 700. Log records  $\langle T_1, C, 700 \rangle$ ,  $\langle T_1, \mathbf{abort} \rangle$  are written out.
- (c) redo ( $T_0$ ) and redo ( $T_1$ ): A and B are set to 950 and 2050 respectively. Then C is set to 600

# CHECKPOINTS

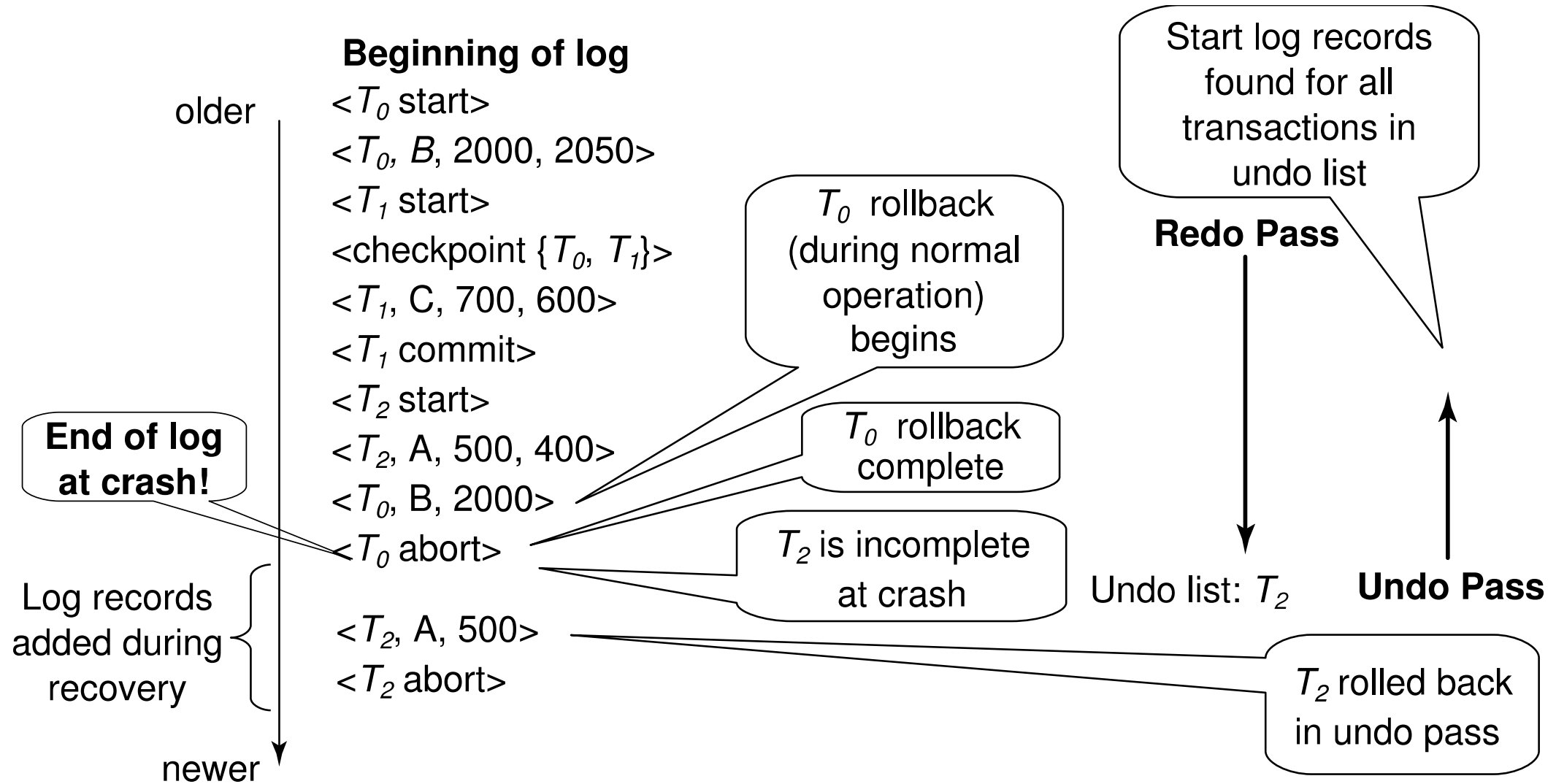
- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**  $L$  > onto stable storage where  $L$  is a list of all transactions active at the time of checkpoint.
  4. All updates are stopped while doing checkpointing

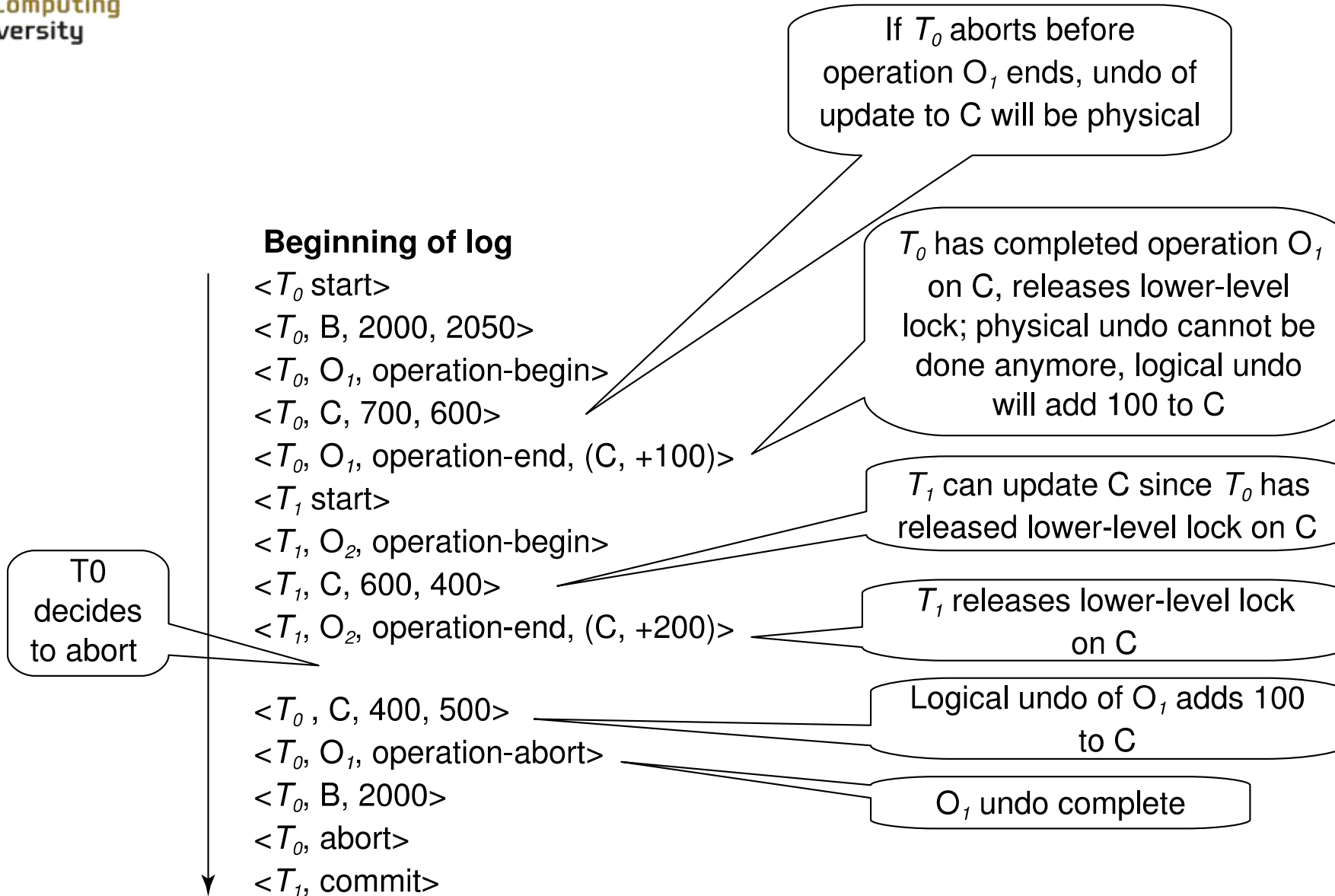


## CHECKPOINTS (CONT.)

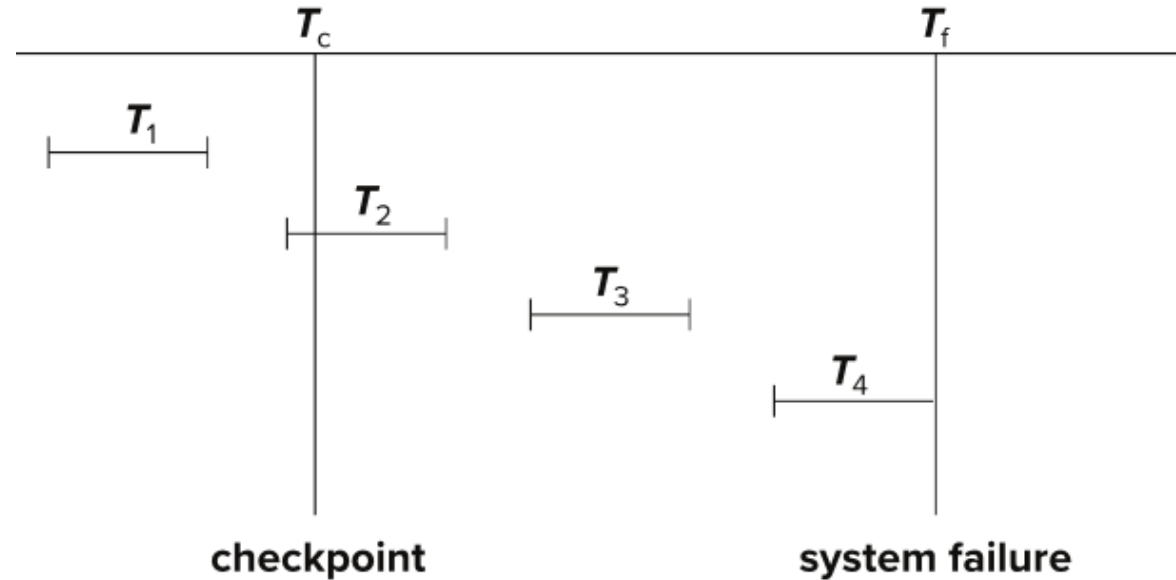
- During recovery we need to consider only the most recent transaction  $T_i$  that started before the checkpoint, and transactions that started after  $T_i$ .
  - Scan backwards from end of log to find the most recent **<checkpoint  $L$ >** record
  - Only transactions that are in  $L$  or started after 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.







# EXAMPLE OF CHECKPOINTS



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





Examine the following **immediate transaction log**.

		A	B	C	D	E	F
1	<T0,START>	900	250	750	900	775	300
2	<T0,A,900,1800>						
3	<T0,B,250,500>						
4	<T1,START>						
5	<T1,C,750,200>						
6	<T2,START>						
7	<T2,E,775,920>						
8	<T0,COMMIT>						
9	<T3,START>						
10	<CHECKPOINT>						
11	<T2,A,1800,600>						
12	<T3,B,500,750>						
13	<T4,START>						
14	<T3,D,900,600>						
15	<T3,ROLLBACK>						
16	<T2,B,750,800>						
17	<T4,START>						
18	<T2,A,600,200>						
19	<T4,F,300,800>						
20	<T4,F,800,900>						
21	<T1,COMMIT>						
	** CRASH **						

Answer the following questions based on the transaction logs:

What are the value of the following variables at the time of crash?

A =

B =

C =

D =

E =

F =

What are the value of the following variables after Recovery?

A =

B =

C =

D =

E =

F =



# REFERENCES

Silberschatz, Korth, and Sudarshan. *Database System Concepts* – 7<sup>th</sup> Edition. McGraw-Hill. 2019.

Slides adapted from Database System Concepts Slide.

Source: <https://www.db-book.com/db7/slides-dir/index.html>

Elmasri, Navathe, “Fundamental of Database Systems”, Seventh Edition, Pearson, 2015.





# ANY QUESTIONS?





# THANK YOU

Database System