Database Recovery

Dr. Bassam Hammo

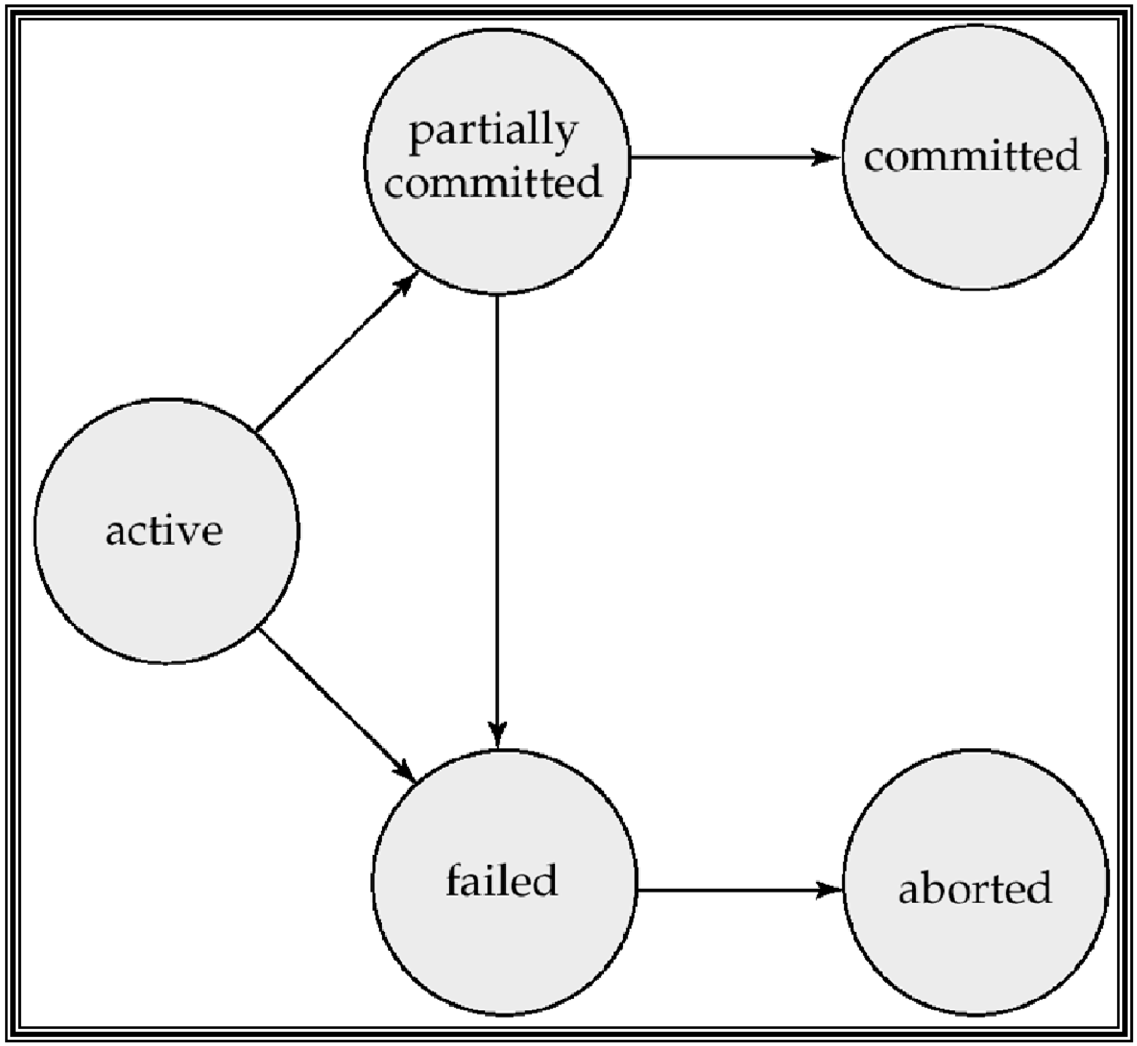
Transaction Concept

- A **transaction** is a *unit* of execution
- Either committed or aborted.
- *After* a transaction, the db must be *consistent*.
  - Consistent – No violation of any constraint.

  For example, if a transaction is supposed to raise the salaries of all employees,

  then the database should guarantee that when the transaction finishes, all salaries should have been raised correctly.

Transaction State



ACID Properties

Each transaction should have:

- **Atomicity.** Either committed or aborted.
- **Consistency.** No violation of any constraint.
- **Isolation.** Concurrent transactions are not aware of each other.
  - Each would think it was the only running transaction
- **Durability.** If the transaction is committed, its changes to the db are permanent.
  - Even if there is a system failure.

Example of Fund Transfer

- Transfer \$50 from account A to B:
  1. **read**(A)
  2. $A = A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B = B + 50$
  6. **write**(B)
- **Consistency** – Assume there is a user constraint that $A + B$ should remain the same. Then the database should ensure this.
- **Atomicity** – If any step fails, then no change should be made to the database.

Example of Fund Transfer (Cont.)

1. **read**(A)
2. $A = A - 50$
3. **write**(A)
4. **read**(B)
5. $B = B + 50$
6. **write**(B)

- **Durability** – once the transaction is complete, the money transfer is permanent.
- **Isolation** – Assume after step 3, another transaction also needs to access A, B. Neither transaction should affect the other.

Recovery Algorithms

- **Recovery algorithms** are techniques to ensure database consistency, transaction atomicity, and durability despite failures.

- **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

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account *A* to account *B*; our goal is either to
  - perform all database modifications made by T_i, or
  - none at all.
- Operations in the transaction
  - Deduct from A
  - Add into B
  - Either one may fail.

Recovery and Atomicity (Cont.)

- We will introduce two recovery methods:
  - **log-based recovery**
  - **shadow-paging**

- We first assume that transactions run serially, that is, one after the other.
- And then address recovery for concurrent transactions.

Log-Based Recovery

- A **log** is kept on stable storage.
  - Contains a sequence of **log records**, described as follows.

- When transaction T_i starts, it registers itself by writing a

<T_i **start**> log record

- Before T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written,
  - V_1 is the value of X before the write
  - V_2 is the value to be written to X.
- When T_i finishes its last statement, the log record <T_i **commit**> is written.
  - Partial commit

Methods of Modifying the Database

- We assume all the log records are written immediately to the disk.
- But as for modifying the database contents, we have:

- Deferred modification.
  - The database simply records all modifications to the log, but defers all the writes to the disk after partial commit.

- Immediate modification.
  - Change the content of the disk immediately (before partial commit).

Deferred Database Modification

- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(X) operation results in a log record <T_i, X,V>, where V is the new value for X.
  - Note: old value is not needed for this scheme
  - The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Example

T_0 :

read (A)

$A = A - 50$

write (A)

read (B)

$B = B + 50$

write (B)

T_1 :

read (C)

$C = C - 100$

write (C)

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

Example With Crashes

T_0 :	T_1 :
read (A)	**read** (C)
$A = A - 50$	$C = C - 100$
write (A)	**write** (C)
read (B)	
$B = B + 50$	
write (B)	

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- Consider the following logs
  In (a), for example, there is a crash before T0 finishes.

Deferred Database Modification

- During the recovery from a crash, a transaction is re-executed if
  - both <T_i **start**> and< T_i**commit**> are present in the log.
- Redoing a transaction T_i sets the value of all data items according to the log records.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- What if there is a crash during the redoing?
  - Say crashes in executing <T0, B, 2050> for (c)?

Deferred Database Modification

- It doesn't matter.
- During recovery from this crash, re-do again.
- Logs are **idempotent.**
- That is, even if the operation is executed multiple times the effect is the same as if it is executed once

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

Immediate Modification – Example

Log	**Update the variable**
< T_0 **start**>	
< T_0, A, 1000, 950>	
< T_o, B, 2000, 2050>	
	A = 950
	B = 2050
< T_0 **commit**>	
< T_1 **start**>	
< T_1, C, 700, 600>	
	C = 600
< T_1 **commit**>	

- Update log record must be written <u>before</u> database item is written.

Immediate Database Modification

- Recovery procedure has two operations instead of one:
  - **undo**(T_i)
    - sets the items updated by T_i to their old values,
    - going backwards from the last log record for T_i
  - **redo**(T_i)
    - sets the items updated by T_i to the new values,
    - going forward from the first log record for T_i

- Both operations must be idempotent

Immediate Database Modification

- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains <T_i **start**>, but not <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both <T_i **start**> and <T_i **commit**>.
- **Undo** operations are performed first, then **redo** operations.

Example with Crashes

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are
set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

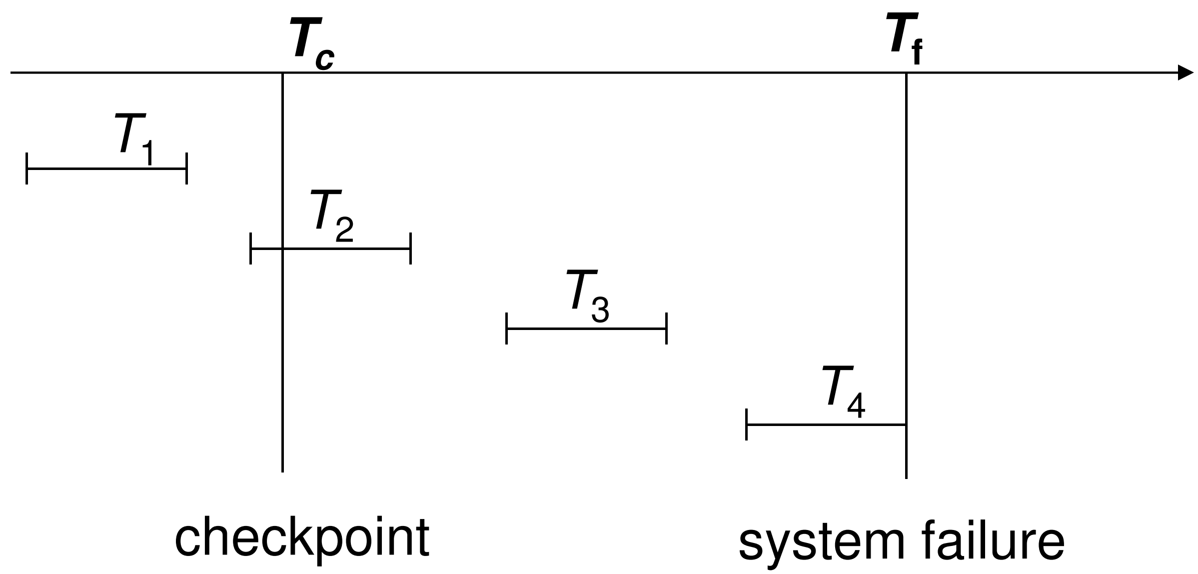
Checkpoints

- In the previous slides, when there are multiple transactions to be executed, we first obtain the logs of all of them, before physically executing the log records.
- Problems:
  - A very long log list.
    - Searching inside the log is time-consuming (e.g., for start/commit records)
  - We might unnecessarily redo transactions multiple times.
    - If a crash happens during redoing.
- Solution: checkpoints

Example

< T_1 **star**t>
< T_1, A, 0, 10>
< T_1 **commit**>
< T_2 **start**>
< T_2, B, 0, 10>
<checkpoint > physically execute the above records
< T_2, C, 0, 10>
< T_2 **commit**>
< T_3 **start**>
< T_3, A, 10, 20>
< T_3, D, 0, 10>
< T_3 **commit**>
< T_4 **start**>
< T_4, A, 20, 30>
failure

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
  - But for T2, redo only the part after the checkpoint.
- T_4 undone

Checkpoints

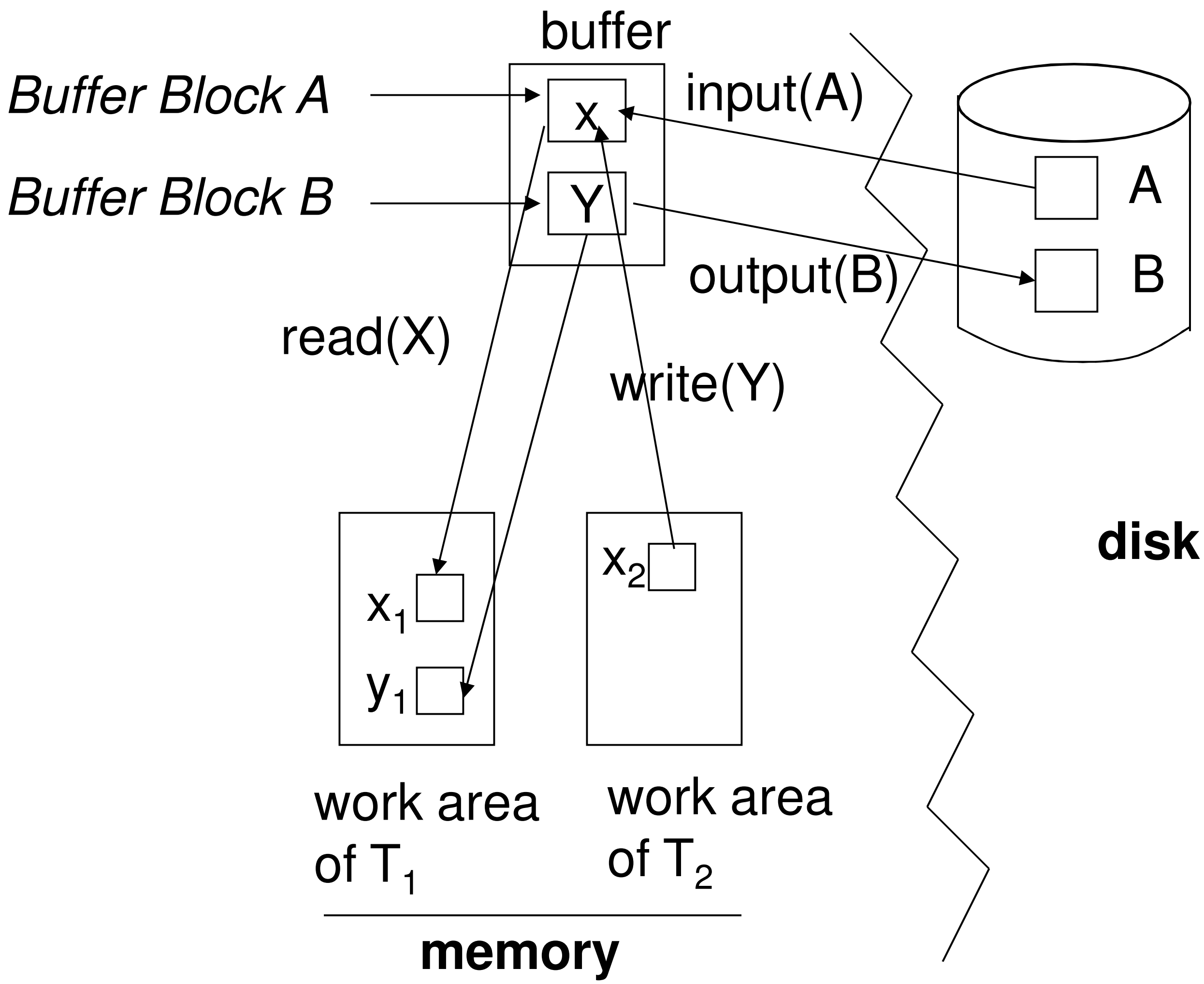
- At each checkpoint, physically execute the log records before it.
- During recovery we need to consider only
  - the most recent transaction that started before the checkpoint
    - E.g., T2 on the previous slide
  - all transactions that started after.
    - E.g., T3, T4

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.

- 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.

Data Access (Cont.)



Data Access (Cont.)

- Two levels of data access
  - **buffer blocks ←→ disk blocks**
  - **transaction work area ←→ buffer blocks**

- **buffer blocks ←→ disk blocks**
  - **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.

Data Access (Cont.)

- **transaction work area ←→ buffer blocks**
  - **read**(X): brings the value of buffered item X to the local variable x_i.
  - **write**(X): assigns the value of local variable x_i to buffered item X.