LECTURE 7
DATABASE RECOVERY
(PART 1/2)

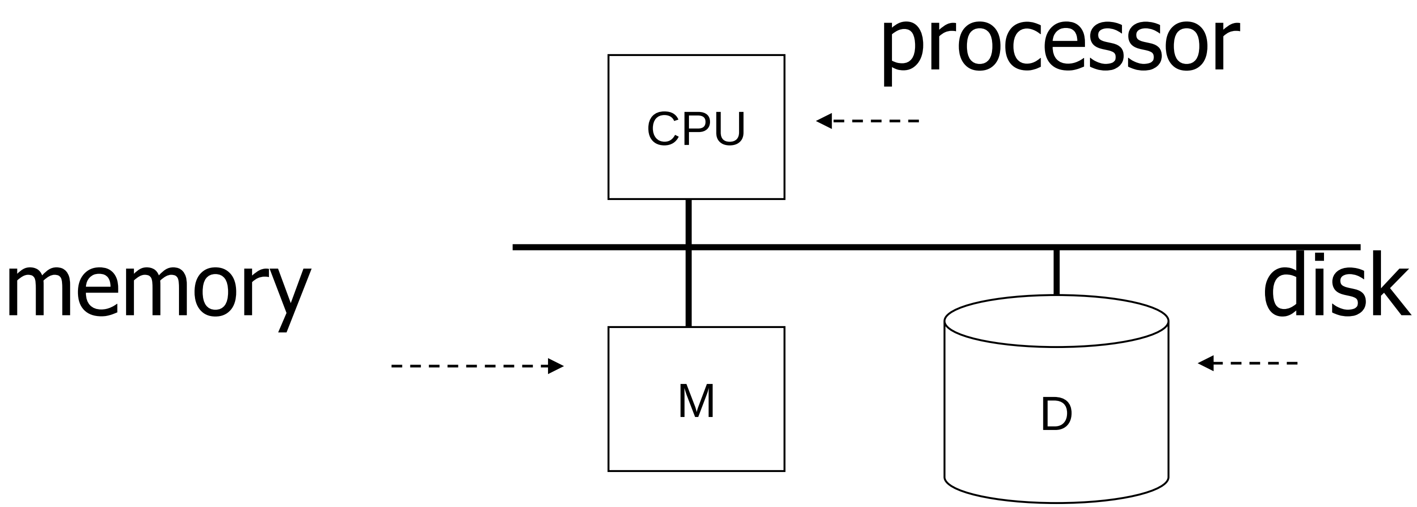
OUTLINES

OUTLINES

17.1 **Issues and Models for Resilient Operation**

17.2 **Undo Logging**

Our failure model



1) Data must be protected in the face of a system failure
logging (undo, redo, undo/redo);
backup (RAID, archiving)
2) Data must not be corrupted simply because several error-free queries or database modifications are being done at once.

Failure Modes

- Erroneous Data entry
  - Some data errors are impossible to detect, some can.
  - Write constraints & triggers to detect data believed to be erroneous
- Media failures
  - Local failures : parity check
  - Disk head crash : RAID; archive; redundant, distributed copies (distributed databases)
- Catastrophic failures
  - archive; redundant, distributed copies
- System failures (soft crash, inconsistent state)
  - **Logging of all database changes in a separate, nonvolatile log, coupled with recovery when necessary**

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Transactions

- *Transaction* = process involving database queries and/or modification.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.

Defining Transactions

- Explicitly

BEGIN TRANSACTION
SQL statement1
SQL statement2
。。。。。
COMMIT

BEGIN TRANSACTION
SQL statement1
SQL statement2
。。。。。
ROLLBACK

- Implicitly
  - Each query or modification statement is a transaction

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - Database modifications are now permanent in the database.

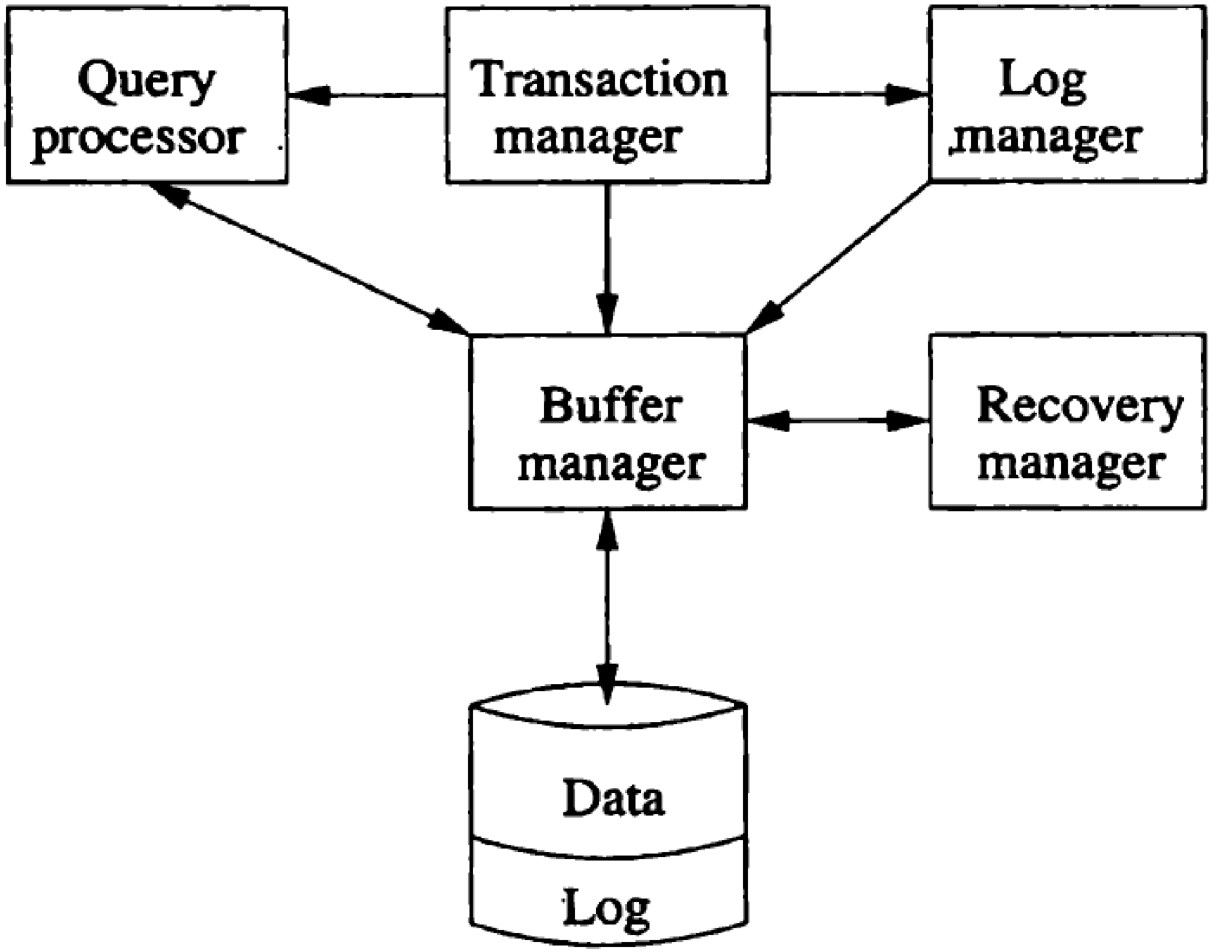
ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.
- Optional: weaker forms of transactions are often supported as well.

Transaction Manager & Log Manager



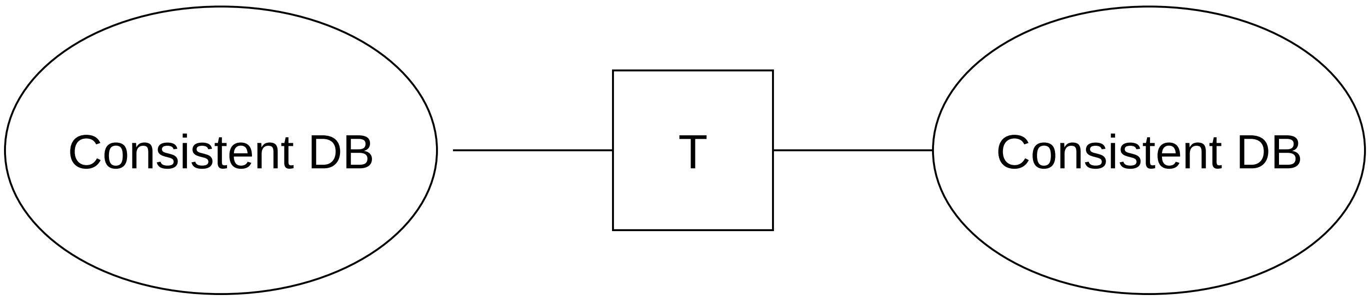
“Elements” for transactions

- Database is composed of “elements.”
- Different database systems use different notions of elements, but they are usually:
  - Relations.
  - Disk blocks or pages.
  - Individual tuples or objects.

Correctness Principle

- State: a DB has a state, which is a value for each of its elements
- Consistent state: certain state satisfies all constraints
- *The Correctness Principle* about transactions: if a transaction executes in the absence of any other transactions or system errors, and it starts with the DB in a consistent state, then the DB is also in a consistent state when the transaction ends.

Transaction: collection of actions that preserve consistency



If T starts with consistent state +

T executes in isolation

$\Rightarrow$ T leaves consistent state

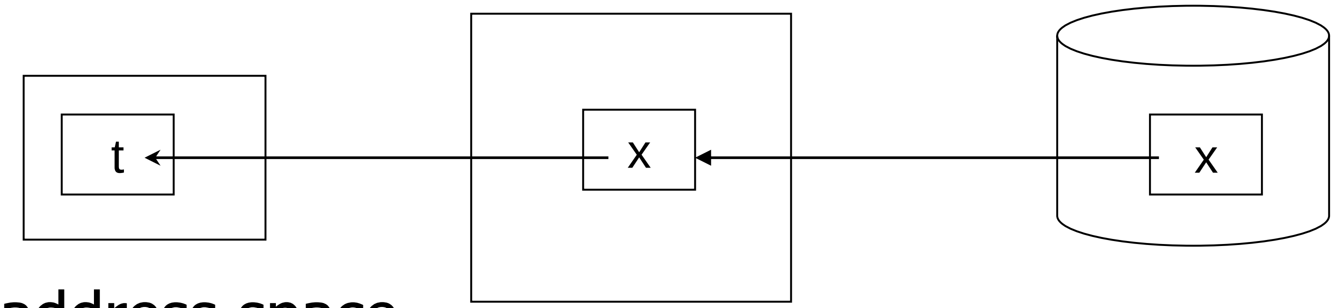
Correctness (informally)

- If we stop running transactions,
  DB left consistent
  1. A transaction is atomic; executed as a whole or not at all. If only part of a transaction executes, then db state may not be consistent
  2. Transactions that execute simultaneously are likely to lead to an inconsistent unless *Concurrency Control*
- Each transaction sees a consistent DB (isolation)

The primitive operations of transaction

Three address spaces

that interact in important ways



Local address space of transaction (Memory)

Buffer(Memory)

Space of Disk

Primitive Operations:

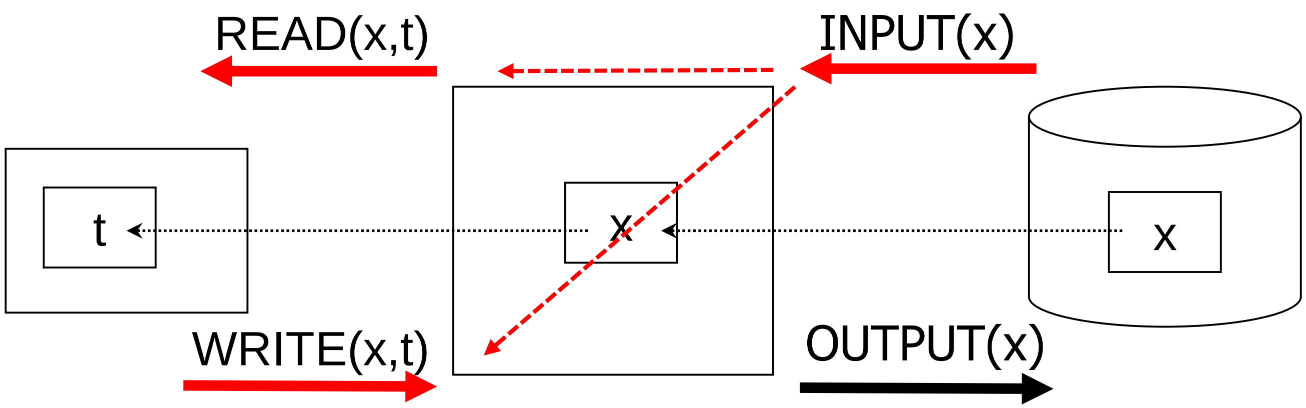
- INPUT(x): block containing x Disk→Buffer/ memory
- OUTPUT(x): block containing x Buffer → disk

- READ(x,t): do input(x) if necessary
  t ← value of x in Buffer
- WRITE(x,t): do output(x) if necessary
  t → value of x in Buffer

A database element is no larger than a single block!!!

The primitive operations of transaction

Three address space



Local address space of transaction (Memory)

Buffer(Memory)

Space of Disk

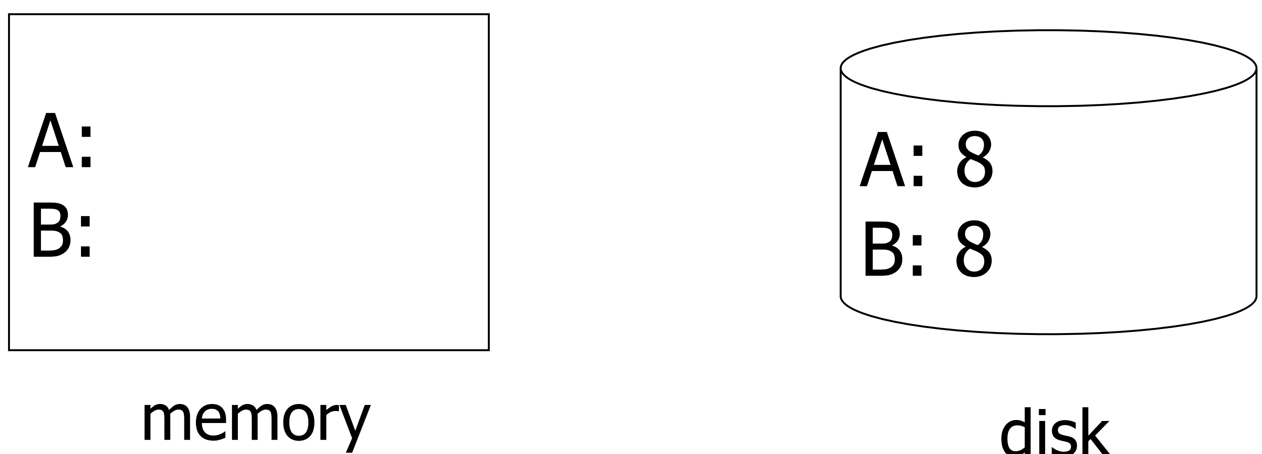
Key problem Unfinished transaction

Example Constraint: A=B

$$T:\ A \leftarrow A \times 2$$

$$B \leftarrow B \times 2$$

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



T: Read (A,t); $t \leftarrow t \times 2$

Write (A,t);

Read (B,t); $t \leftarrow t \times 2$

Write (B,t);

Output (A);

Output (B);

A: 8

B:

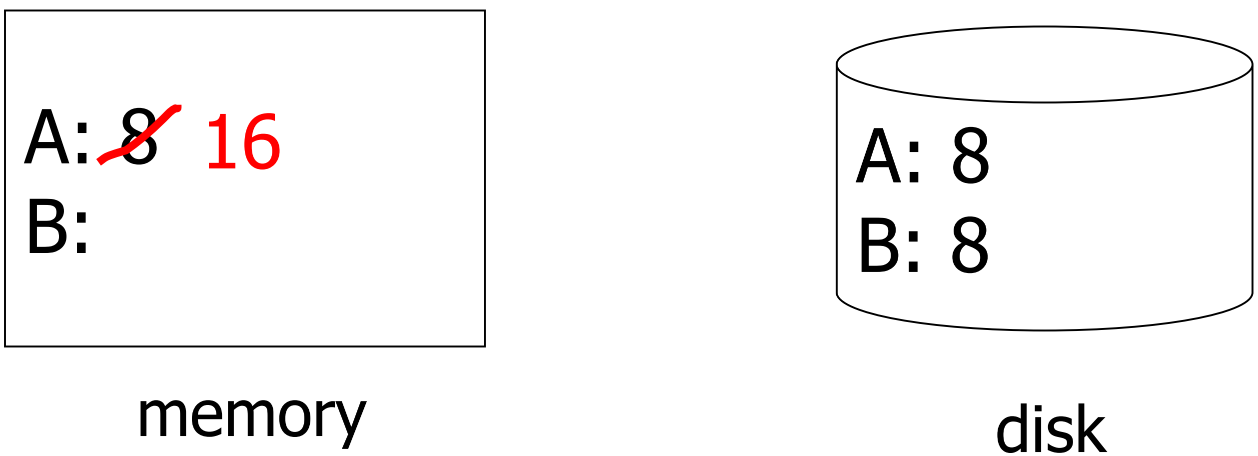
memory

A: 8

B: 8

disk

T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

A: ~~8~~ 16
B: ~~8~~ 16

memory

A: 8
B: 8

disk

T_1: Read (A,t); $t \leftarrow t\times 2$
Write (A,t);
Read (B,t); $t \leftarrow t\times 2$
Write (B,t);
Output (A);
Output (B);

failure!

A: ~~8~~ 16
B: ~~8~~ 16

memory

A: ~~8~~ 16
B: 8

disk

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

Action	*t*	Mem *A*	Mem *B*	Disk *A*	Disk *B*
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

- Need <u>atomicity:</u> execute all actions of a transaction or none at all

- One solution: undo logging

OUTLINES

Undo logging

- **A** log is a file of ***log*** records, each telling something about what some transaction has done.
- Imagine the log as a file opened for appending only.
- Undo logging—makes repairs to the DB state by undoing the effects of transactions that may not completed before the crash.

Log Records

- Several forms of log records
  - **<START *T*>**: indicates that transaction *T* has begun
  - <**COMMIT *T***>*:* Transaction ***T*** has completed successfully and will make no more changes to database elements
  - <**ABORT *T***>*.* Transaction *T* could not complete successfully

Undo log

- Only *update record*, **<*T*, X, *v*>**
  - transaction *T* has changed database element ***X*** and its former value was *v*.
- The change reflected by an update record normally occurs in memory, not disk
  - the log record is a response to a WRITE action, not an OUTPUT action.

Notice

- An undo log does not record the new value of a database element
  - Only the old value

- The only thing the recovery manager will do
  - is to cancel the possible effect of a transaction on disk
  - by restoring the old value

The Undo-Logging Rules

- ***U1:*** If transaction ***T*** modifies database element *X*, then the log record of the form **<T,** ***X*****,** ***v*****>** must be written to disk *before* the new value of *X* is written to disk.

Write disk：first undo-log then data

- ***U2:*** If a transaction commits, then its **COMMIT** log record must be written to disk only *after* all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

COMMIT：first data then undo-log commit

The Undo-Logging Rules

- To summarize rules *U1* and *U2*

  material associated with one transaction must be written to disk in the following order

  a) The log records indicating changed database elements

  b) The changed database elements themselves

  c) The commit log record

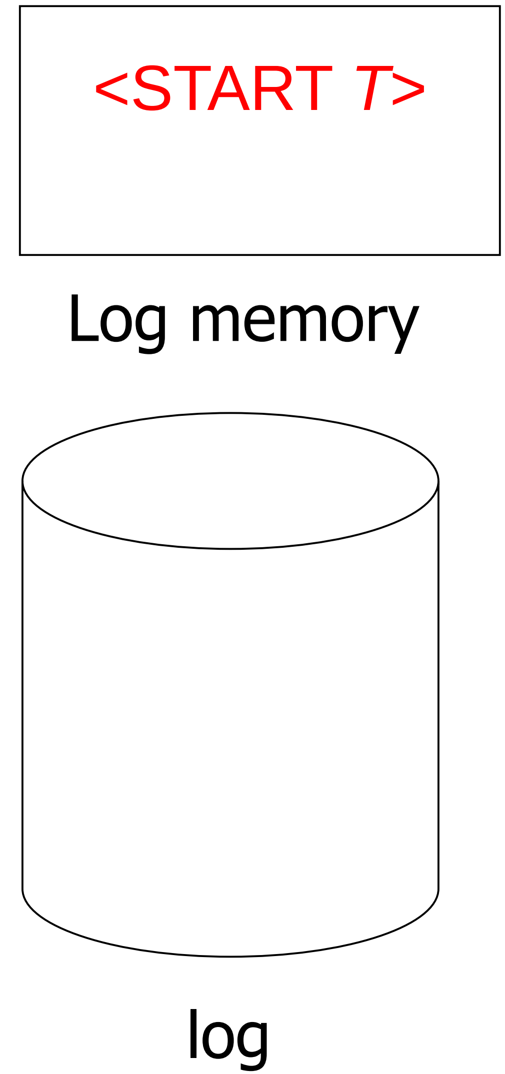
  The order of a) and b) applies to each database element individually

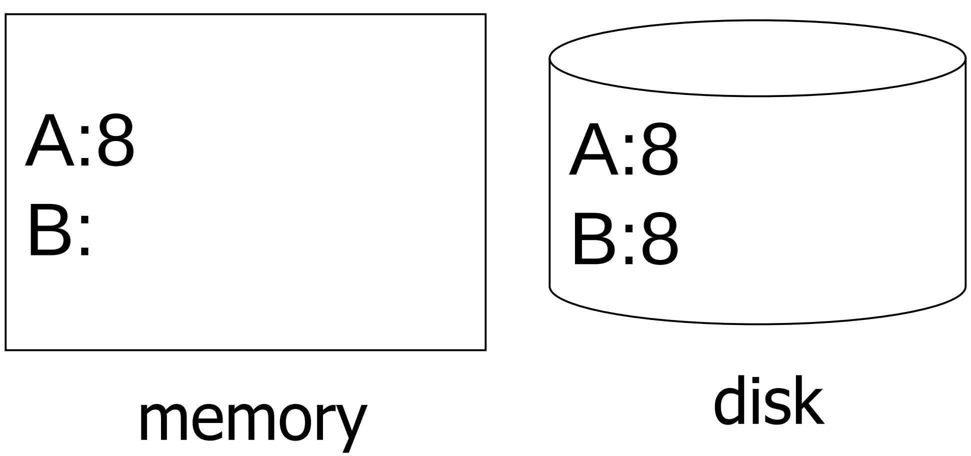
Undo logging rules

- In order to force log records to disk
  - The log manager needs a flush-log command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk
  - FLUSH LOG
- The transaction manager needs to have a way to tell the buffer manager
  - To perform an OUTPUT action on a database element

Undo logging

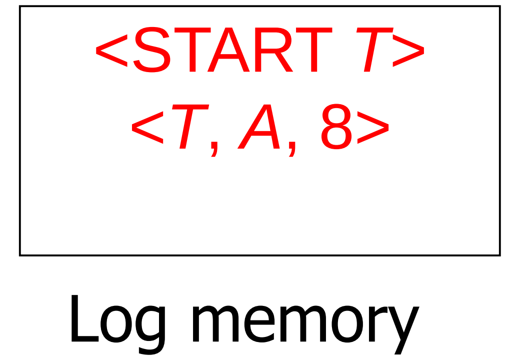
T: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

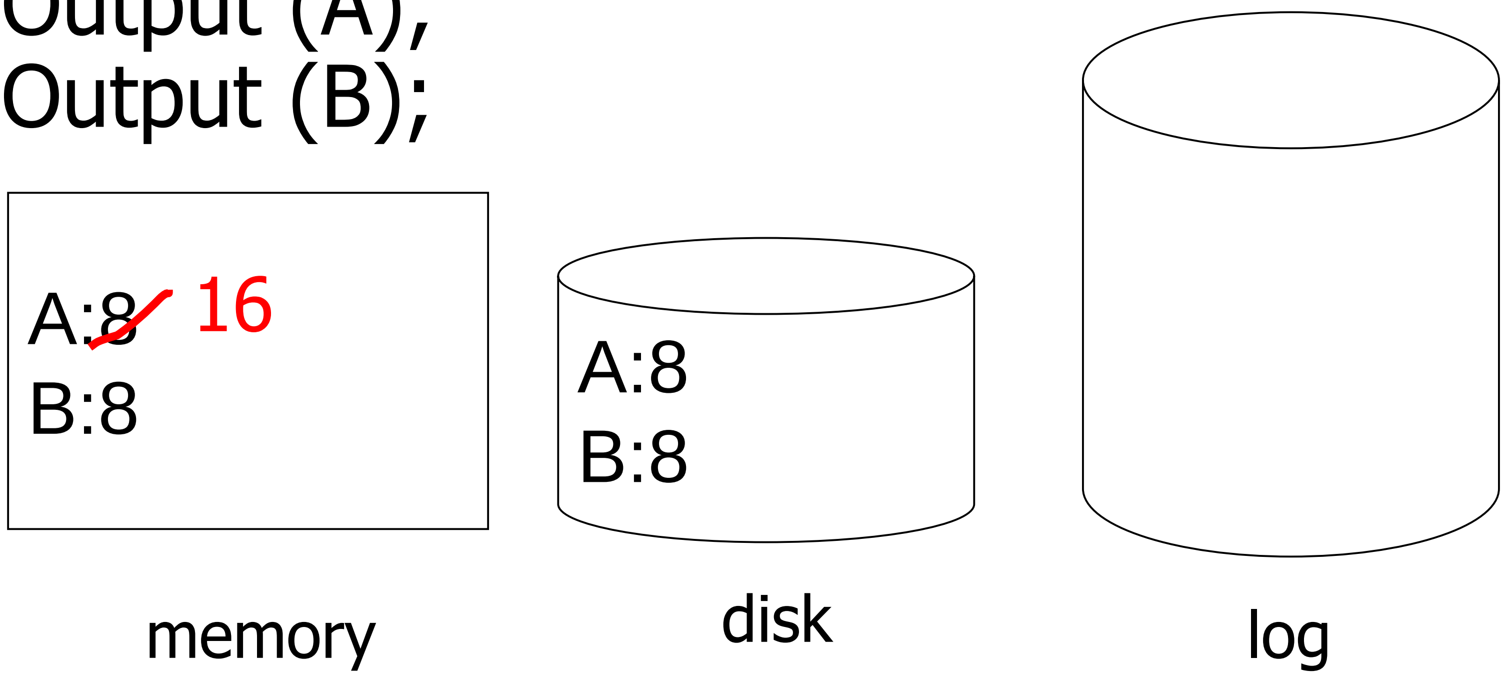




Undo logging

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



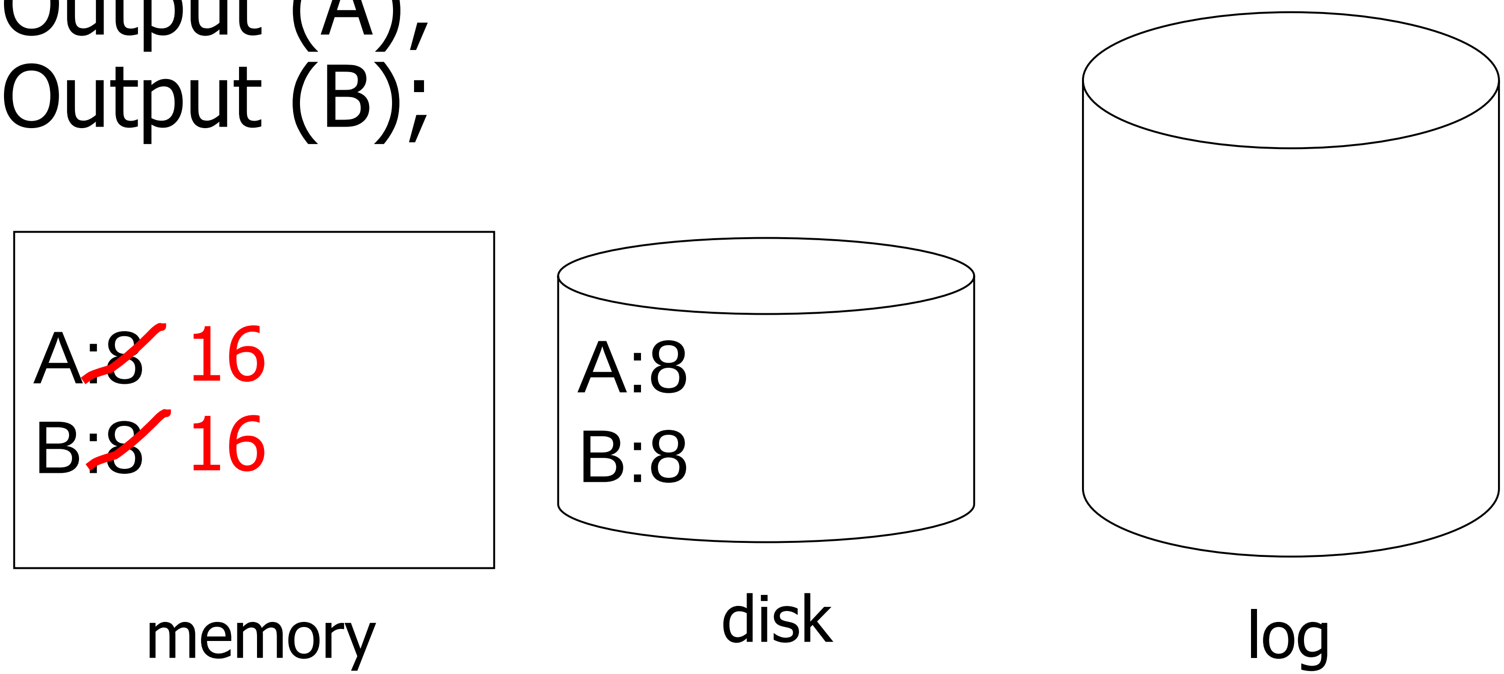


Undo logging

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

<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>

Log memory



Undo logging

T: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t); (flush log)
Output (A);
Output (B);

<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>

Log memory

A:~~8~~ 16
B:~~8~~ 16

memory

A:8
B:8

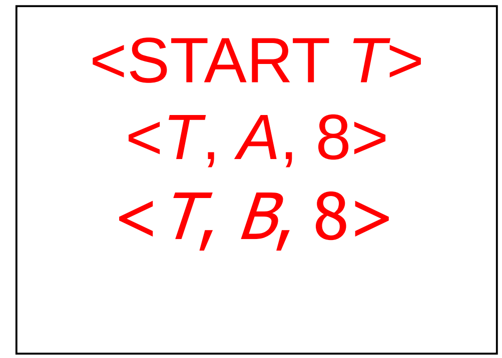
disk

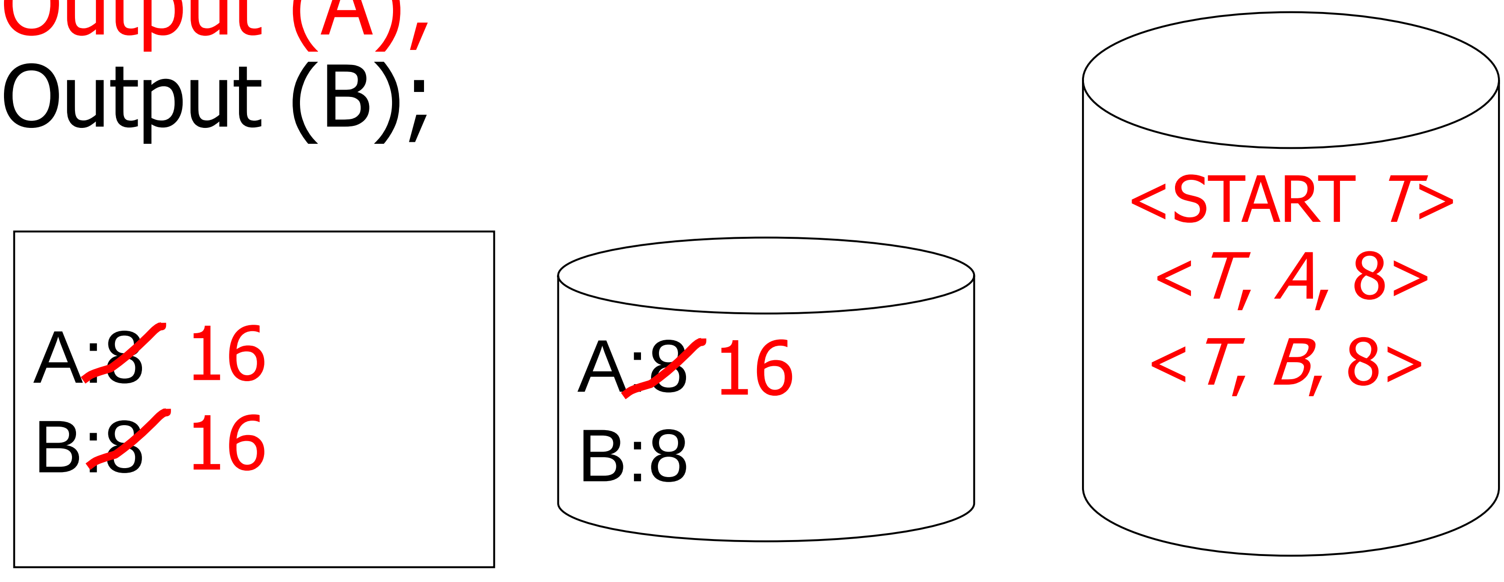
<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>

log

Undo logging

T: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t); (flush log)
Output (A);
Output (B);





Undo logging

T: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t); (flush log)
Output (A);
Output (B);

<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>
<COMMIT *T*>

Log memory

A:~~8~~ 16
B:~~8~~ 16

memory

A:~~8~~ 16
B:~~8~~ 16

disk

<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>

log

Undo logging

T: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t); (flush log)
Output (A);
Output (B); (flush log)

<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>
<COMMIT *T*>

Log memory

A:~~8~~ 16
B:~~8~~ 16

memory

A:~~8~~ 16
B:~~8~~ 16

disk

<START *T*>
<*T*, *A*, 8>
<*T*, *B*, 8>
<COMMIT *T*>

log

Undo logging

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

Step	Action	*t*	M-*A*	M-*B*	D-*A*	D-*B*	Log
1)							<START *T*>
2)	`READ(A,t)`	8	8		8	8	
3)	`t := t*2`	16	8		8	8	
4)	`WRITE(A,t)`	16	16		8	8	<*T*, *A*, 8>
5)	`READ(B,t)`	8	16	8	8	8	
6)	`t := t*2`	16	16	8	8	8	
7)	`WRITE(B,t)`	16	16	16	8	8	<*T*, *B*, 8>
8)	`FLUSH LOG`						
9)	`OUTPUT(A)`	16	16	16	16	8	
10)	`OUTPUT(B)`	16	16	16	16	16	
11)							<COMMIT *T*>
12)	`FLUSH LOG`						

Recovery Using Undo Logging

- Here only the simplest form of recovery
  - One that looks at the entire log
- To divide the transactions into
  - Committed transactions
    - <COMMIT *T*>
    - By undo rule *U*2, all changes made by T were previously written to disk
    - Consistent state
  - Uncommitted transactions

Recovery rules: Undo logging

Scan the undo log from the end:

Incomplete transaction must be undone!

(1) Let S = set of transactions with

<START T_i> in log, but no

<COMMIT T_i> or <ABORT T_i> record in log

(2) For each <T_i, X, v> in log,

in reverse order (latest $\rightarrow$ earliest) do:

- If $T_i \in S$ then $\begin{cases} \text{WRITE } (X, v) \\ \text{OUTPUT } (X) \end{cases}$

(3) For each $T_i \in S$ do

- Write <ABORT T_i> to log

Example 17.3

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	$<T, A, 8>$
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	$<T, B, 8>$
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

1. **The crash occurs after step (12).**
2. **The crash occurs between steps (11) and (12).**
3. **The crash occurs between steps (10) and (11).**
4. **The crash occurs between steps (8) and (10).**
5. **The crash occurs prior to step (8).**

Checkpointing

- To avoid examining the entire log
  - Can we delete the log prior to a COMMIT?
  - No! log records pertaining to some other active transaction *T* might be lost

- Solution to this problem
  - To checkpoint the log periodically

Checkpointing

- In a simple checkpoint,
  1. Stop accepting new transactions.
  2. Wait until all currently active transactions commit or abort and have written a COMMIT or ABORT record on the log.
  3. Flush the log to disk.
  4. Write a log record <CKPT>, and flush the log again.
  5. Resume accepting transactions.

Checkpointing

- The effect of a checkpoint
  - Any transaction that executed prior to the checkpoint has finished
  - During a recovery, scan the log from the end, when find a <CKPT> record, all the incomplete transactions have been seen
  - No need to scan prior to the <CKPT>,
  - In fact the log before that point can be deleted or overwritten safely.

Checkpointing

- Example

<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>

Do a checkpoint

<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>
<$T_2, C, 15$>
<$T_1, D, 20$>
<COMMIT T_1>
<COMMIT T_2>
<CKPT>
<START T_3>
<$T_3, E, 25$>
<$T_3, F, 30$>

Suppose a crash occurs at this point

Nonquiescent checkpointing

- Problem with the simple checkpointing
  - Must shut down the system while the checkpoint is being made
- Solution
  - Nonquiescent checkpointing
    - Allows new transactions to enter the system during the checkpoint

Nonquiescent checkpointing

- Nonquiescent checkpointing
  1. Write a log record
     <START CKPT (T_1, …, T_K)>
     and flush the log.
  2. Wait until all of T_1, …, T_K commit or abort, but do not prohibit other transactions from starting
  3. When all of T_1, …, T_K have completed, write a log record <END CKPT> and flush the log

Nonquiescent checkpointing

- Recovery
  - Two cases
    1. If we first meet an <END CKPT>, then we know that all incomplete transactions began after the previous <START CKPT (T_1, …, T_K)>
       - Scan backwards as far as <START CKPT (T_1, …, T_K)>

Nonquiescent checkpointing

A general rule:

Once an <END CKPT> has been written to disk, we can delete the log prior to the previous <START CKPT …>

- Recovery
  - Two cases
    2. If we first meet an <START CKPT (T_1, …, T_K)>, then the crash occurred during the checkpoint

<START CKPT (T_1, T_2)>

<END CKPT>

<START CKPT (T_3, T_4, T_5)>

Crash

Recovery

Nonquiescent checkpointing

- Example

$<\mathrm{START}\ T_1>$
$<T_1, A, 5>$
$<\mathrm{START}\ T_2>$
$<T_2, B, 10>$

Do a nonquiescent checkpoint

$<\mathrm{START}\ T_1>$
$<T_1, A, 5>$
$<\mathrm{START}\ T_2>$
$<T_2, B, 10>$
$<\mathrm{START\ CKPT}\ (T_1, T_2)>$
$<T_2, C, 15>$
$<\mathrm{START}\ T_3>$
$<T_1, D, 20>$
$<\mathrm{COMMIT}\ T_1>$
$<T_3, E, 25>$
$<\mathrm{COMMIT}\ T_2>$
$<\mathrm{END\ CKPT}>$
$<T_3, F, 30>$

Suppose a crash occurs at this point

Nonquiescent checkpointing

- Example

<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>

Do a nonquiescent checkpoint →

<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>
<START CKPT (T_1, T_2)>
<$T_2, C, 15$>
<START T_3>
<$T_1, D, 20$>
<COMMIT T_1>
<$T_3, E, 25$>

Suppose a crash occurs at this point (during the checkpoint)