

Recovery Algorithms

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures

Focus of this chapter

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

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.

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 .

We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

Transaction transfers data items between system buffer blocks and its private work-area using the following operations :

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.

both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.

Transactions

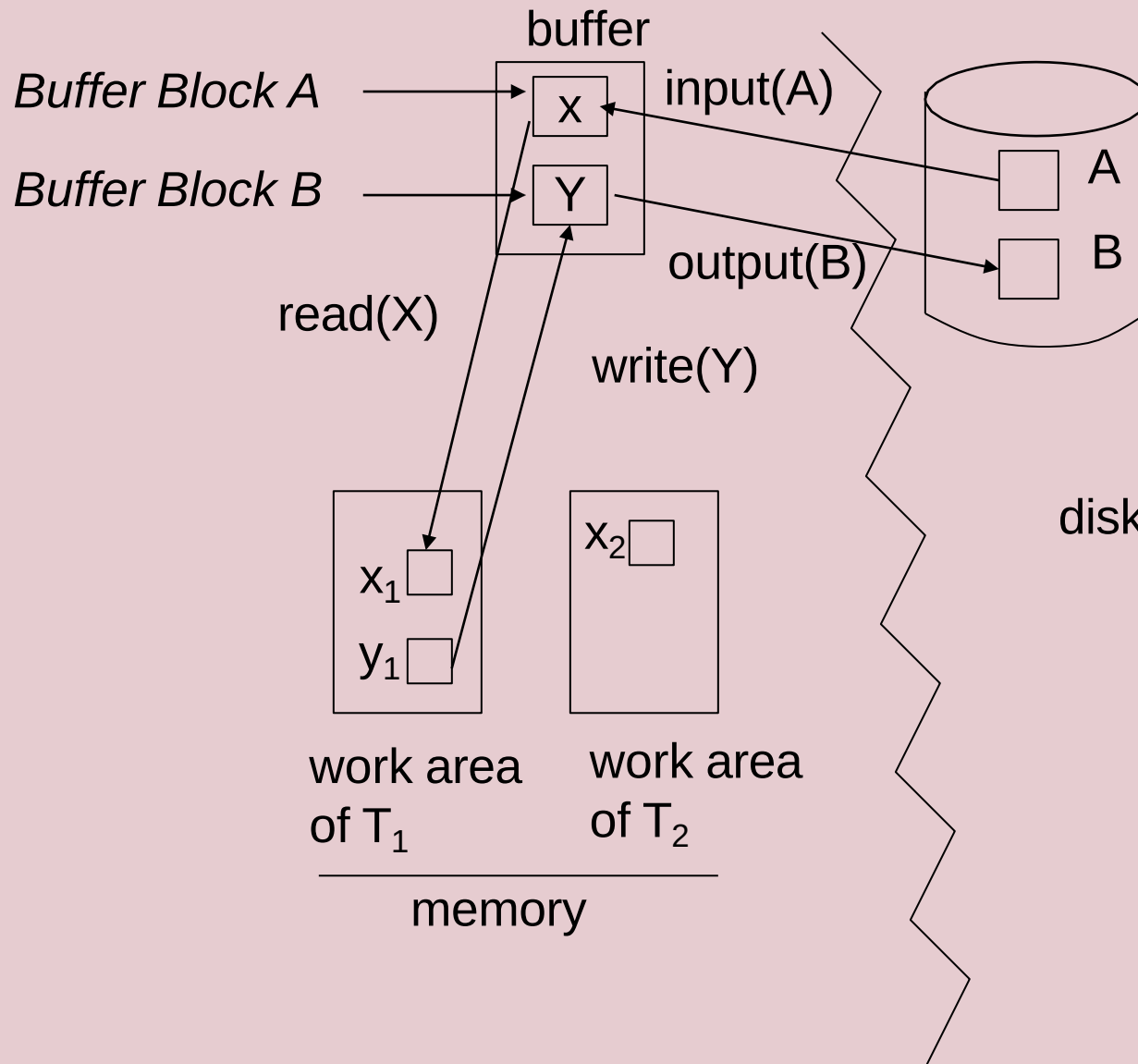
Perform **read**(X) while accessing X for the first time;

All subsequent accesses are to the local copy.

After last access, transaction executes **write**(X).

output(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access



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 ; goal is either to perform all database modifications made by T_i or none at all.

Several output operations may be required for T_i (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.

Recovery and Atomicity (Cont.)

To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.

We study a approach:

log-based recovery

We assume (initially) that transactions run serially, that is, one after the other.

Log-Based Recovery

A **log** is kept on stable storage.

The log is a sequence of **log records**, and maintains a record of update activities on the database.

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, and V_2 is the value to be written to X .

Log record notes that T_i has performed a write on data item X_j . X_j had value V_1 before the write, and will have value V_2 after the write.

When T_i finishes its last statement, the log record $\langle T_i \text{ commit} \rangle$ is written.

We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches using logs

- Deferred database modification

- Immediate database modification

Deferred Database Modification

The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.

Assume that transactions execute serially

Transaction starts by writing $\langle T_i, \text{start} \rangle$ record to log.

A **write**(X) operation results in a log record $\langle T_i, X, V \rangle$ being written, 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, $\langle T_i, \text{commit} \rangle$ is written to the log

Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

During recovery after a crash, a transaction needs to be redone if and only if both $\langle T_i \text{ start} \rangle$ and $\langle T_i \text{ commit} \rangle$ are there in the log.

Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.

Crashes can occur while

the transaction is executing the original updates, or
while recovery action is being taken

example transactions T_0 and T_1 (T_0 executes before T_1):

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)

Deferred Database Modification (Cont.)

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

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

If log on stable storage at time of crash is as in case:

- (a) No redo actions need to be taken
- (b) redo(T_0) must be performed since $\langle T_0 \text{ commit} \rangle$ is present
- (c) redo(T_0) must be performed followed by redo(T_1) since $\langle T_0 \text{ commit} \rangle$ and $\langle T_1 \text{ commit} \rangle$ are present

Immediate Database Modification

The immediate database modification scheme allows database updates of an uncommitted transaction to be made as the writes are issued since undoing may be needed, update logs must have both old value and new value

Update log record must be written before database item is written

We assume that the log record is output directly to stable storage

Can be extended to postpone log record output, so long as prior to execution of an output(B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage

Output of updated blocks 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.

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, x_1 600>		
	C = 600	
		B_B, B_C
< T_1 commit>		
		B_A

Note: B_x denotes block containing X.

Immediate Database Modification (Cont.)

Recovery procedure has two operations instead of one:

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

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

Both operations must be **idempotent**

That is, even if the operation is executed multiple times the effect is the same as if it is executed once

- ▶ Needed since operations may get re-executed during recovery

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 the record $\langle T_i \text{ commit} \rangle$.

Transaction T_i needs to be redone if the log contains both the record $\langle T_i \text{ start} \rangle$ and the record $\langle T_i \text{ commit} \rangle$.

Undo operations are performed first, then redo operations.

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：<https://d.book118.com/498052011012006042>