Recovery

Recoverable

- Transactions may be aborted due to logical failure. e.g. deadlock
- Recoverability is required to ensure that aborting a transaction does not change the semantics of committed transaction's operations.
- Example
  - $Write_1(x,2)$; $Read_2(x)$; $Write_2(y,3)$; $Commit_2$
  - Not recoverable.
  - T_2 has committed before T_1 commits.
  - The problem is: what can we do if T_1 abort?
  - Delaying the commitment of T_2 can avoid this problem
- A schedule *H* is called *recoverable* (RC) if, whenever T_i reads from T_j. ($i \neq j$) in *H* and $c_i \in H$, $c_j < c_i$.
- Intuitively, a history is recoverable if each transaction commits after the commitment of all transactions (other than itself) from which it reads.

Avoiding Cascading Aborts

- Even for recoverable execution, aborting a transaction may trigger further abortions, a phenomenon called *cascading abort*.
- Example
  - $Write_1(x,2)$; $Read_2(x)$; $Write_2(y,3)$; $Abort_1$
  - T_2 must abort because if it ever committed, the execution would no longer valid.
- A schedule *H* avoids cascading aborts (ACA) if, whenever T_i reads x from T_j $(i \neq j)$, $c_j < r_i[x]$.
- That is, a transaction may read only those values that are written by committed transactions or by itself.

Strict Executions

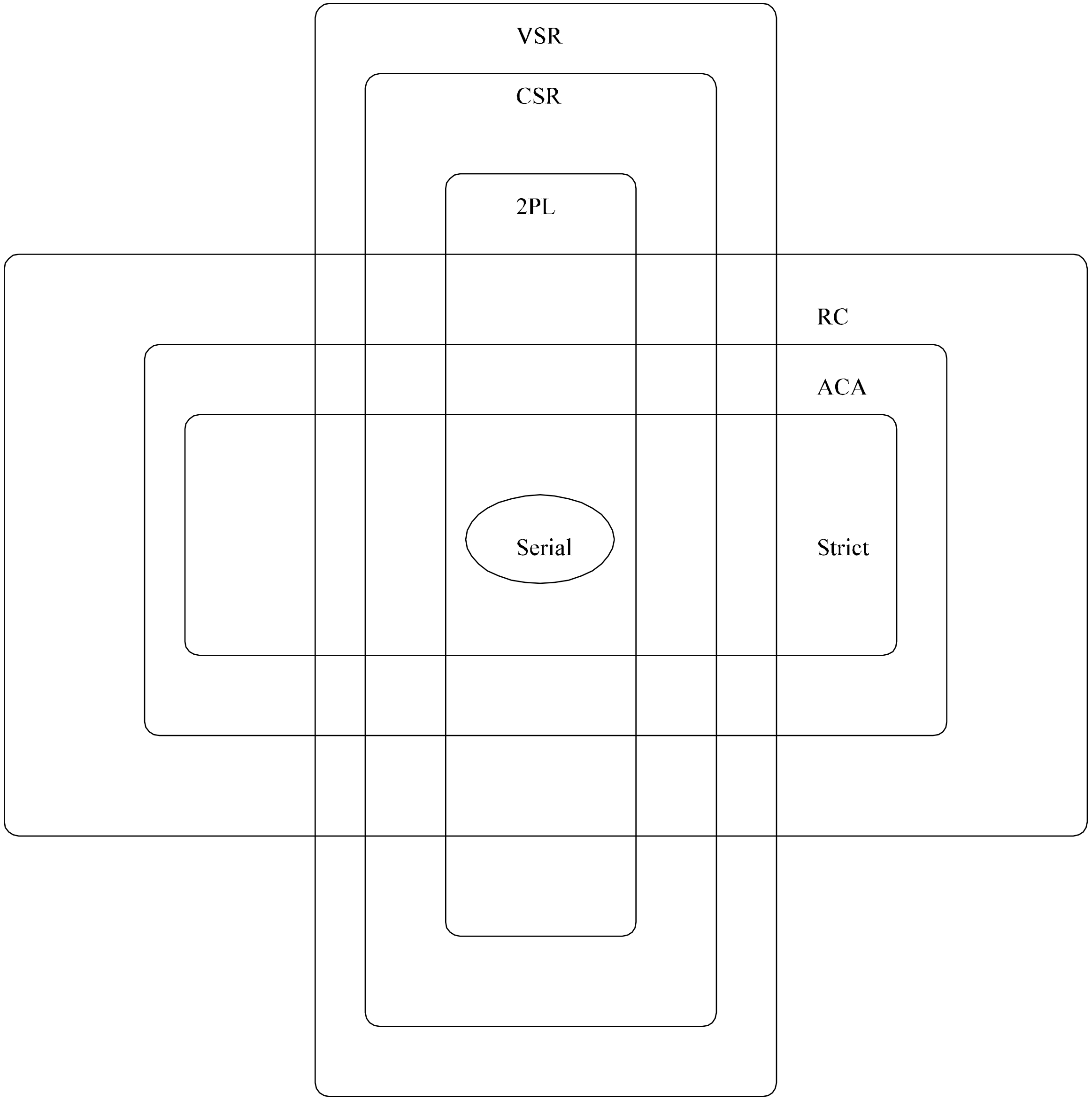
- Avoiding cascading aborts is not always enough from the practical point of view.
- $Write_1(x,1)$; $Write_1(y,3)$; $Write_2(y,1)$; $Commit_1$; $read_2(x)$; $Aborts_2\$$
- If we erase the operations from T_2, the resulting execution should be
  - $Write_1(x,1)$; $Write_1(y,3)$; $Commit_1$
- The value of *y* should be 3.
- The value should be restored for *y* when T_2 is aborted.
- Implement abort by restoring the before images of all Writes of a transaction.
- The before image of a *Write(x,val)* operation in an execution is the value of *x* just before this write operation.

- The following example illustrates the problem
  - $Write_1(x,2)$; $Write_2(x,3)$; $Abort_1$
  - Assume the initial value of *x* is 1.

    i.e., the before image of $Write_1(x,2)$ is 1.
  - Blindly restoring the before image will lead to a wrong result.
- Another example (assume the initial value of *x* is 1)
  - $Write_1(x,2)$; $Write_2(x,3)$; $Abort_1$; $Abort_2$
  - The before image of $Write_2(x,3)$ is 2.
  - After $Write_2(x,3)$ has been undone, the value of *x* should be 1.
- We can avoid these problems by requiring that the execution of a Write(x,val) be delayed until all transactions that have previously written *x* are either committed or aborted.
- A schedule *H* is strict (ST) if whenever $w_j[x] < o_i[x]$ ($i \neq j$), either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$.
- This property can be enforced by using the strict two-phase locking protocol (locks are released at the end of transactions), i.e., Strict 2PL guarantees both conflict serializability and strict execution.

- Examples
  - T1 = w1[x] w1[y] w1[z] c1
  - T2 = r2[u] w2[x] r2[y] w2[y] c2
  - H1 = w1[x] w1[y] r2[u] w2[x] r2[y] w2[y] c2 w1[z] c1
  - H2 = w1[x] w1[y] r2[u] w2[x] r2[y] w2[y] w1[z] c1 c2
  - H3 = w1[x] w1[y] r2[u] w2[x] w1[z] c1 r2[y] w2[y] c2
  - H4 = w1[x] w1[y] r2[u] w1[z] c1 w2[x] r2[y] w2[y] c2

  - H1 is not RC, because T2 reads *y* from T1, but c2 < c1.
  - H2 is RC but not ACA, because T2 has read y from T1 before T1 commits.
  - H3 is ACA but not ST, because T2 has overwritten the value written into x by T1 before T1 terminates
  - H4 is strict.


VSR
CSR
2PL
RC
ACA
Serial
Strict

Transaction Execution

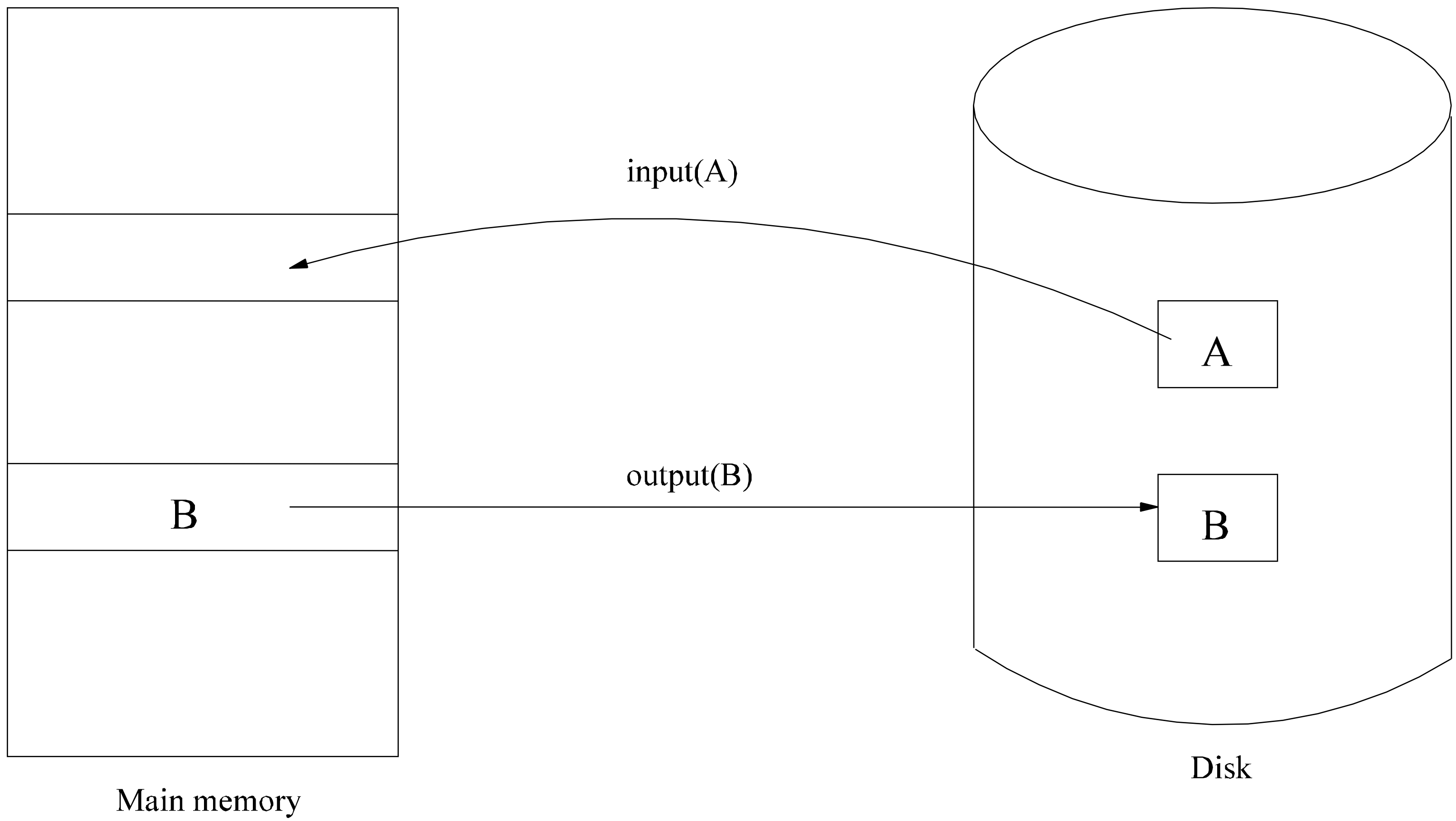
- Assume that strict two-phase locking protocol is used.
  - Locks are released at the end of a transaction.
  - If a transaction T_i writes a data item x, no transaction can read or write x before T_i commits or aborts.
  - When a transaction is aborted, the effect of a write operation can be removed by restoring the before image (value of the data item just before the write operation is executed).
  - Recoverable, Avoiding Cascading Aborts, and Strict Execution are guaranteed.

Crash Recovery

- Storage Types
  - Volatile storage: Does not survive system crashes. e.g. main memory, cache memory.
  - Nonvolatile storage: Survives system crashes. e.g. disk, tapes.
  - Stable storage: never lost. e.g. implemented by replication.

- Failure Types
  - Logical errors: cannot continue execution because of internal conditions. e.g. bad input, data not found, overflow, resource limit exceeded.
  - System errors: the system has entered an undesirable state. e.g. deadlock.
  - System crash: hardware malfunctions, causing the loss of the content of volatile storage. e.g. power failure.
  - Disk failure: disk block loses its content. e.g. head crash, failure during a data transfer operation.

- Storage Hierarchy
  - The database system resides in nonvolatile storage (disk).
  - Data transfer are in terms of block (page).
  - Physical block: block residing on the disk.
  - Buffer block: blocks residing temporarily in main memory.
  - Block movements between disk and main memory are initiated through:
    - input(X): From disk to memory
    - output(X): From memory to disk.


input(A)
A
output(B)
B
B
Main memory
Disk

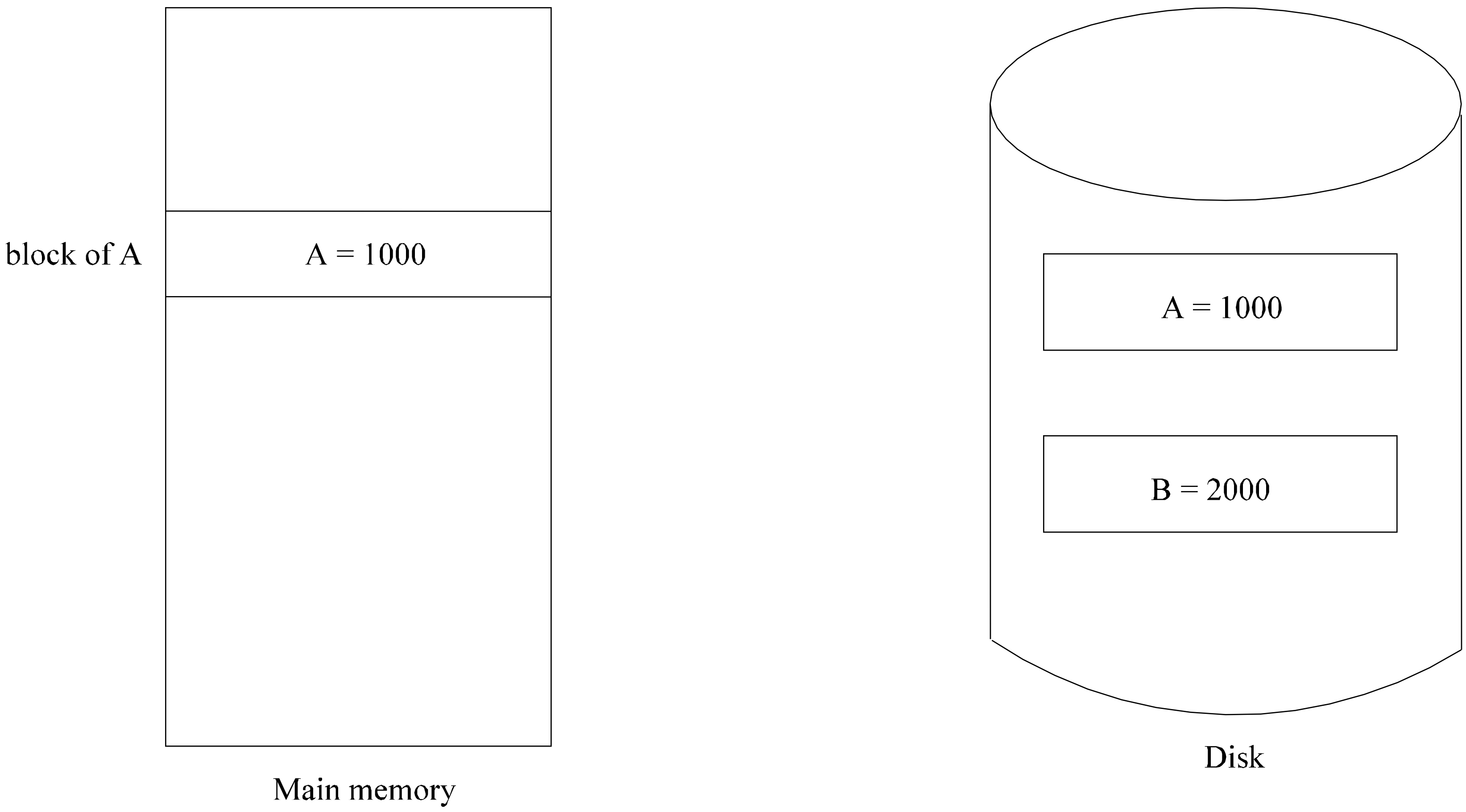
- Transactions interact with the database:
  - Data in database ↔ program variables.
    - read(X,x_i): assigns the value of data item X to the local variable x_i.
      - Issue input(X), if X is not in the buffer.
      - Assign the value of X to x_i.
    - write(X,x_i): assigns the value of local variable x_i to data item X.
      - Issue input(X), if X is not in the buffer.
      - Assign the value of x_i to X in the buffer block for X.

- Not specifically require the transfer of a block from buffer to disk.
- A buffer block is eventually written out to the disk either:
  - Buffer manager needs the memory space.
  - Force-output

Examples

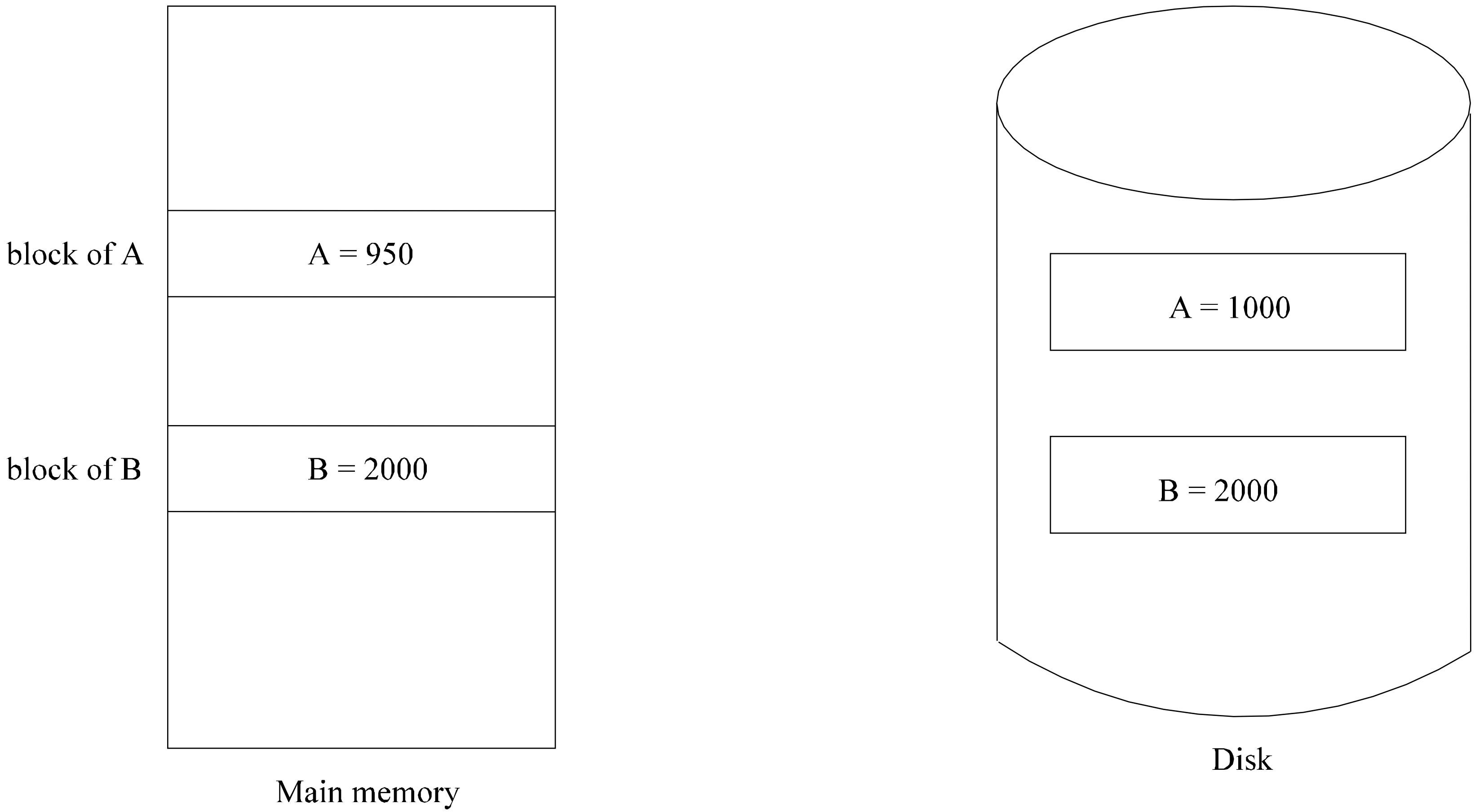
```
read(A,a1)
a1 := a1 - 50
write(A,a1)
read(B,b1)
b1 := b1 + 50
write(B,b1)
```

- The consistency constraint is that the sum of A and B is unchanged.
- Initial values of A and B are $1000 and $2000
- Main memory contains the buffer block A.


block of A
A = 1000
Main memory
A = 1000
B = 2000
Disk

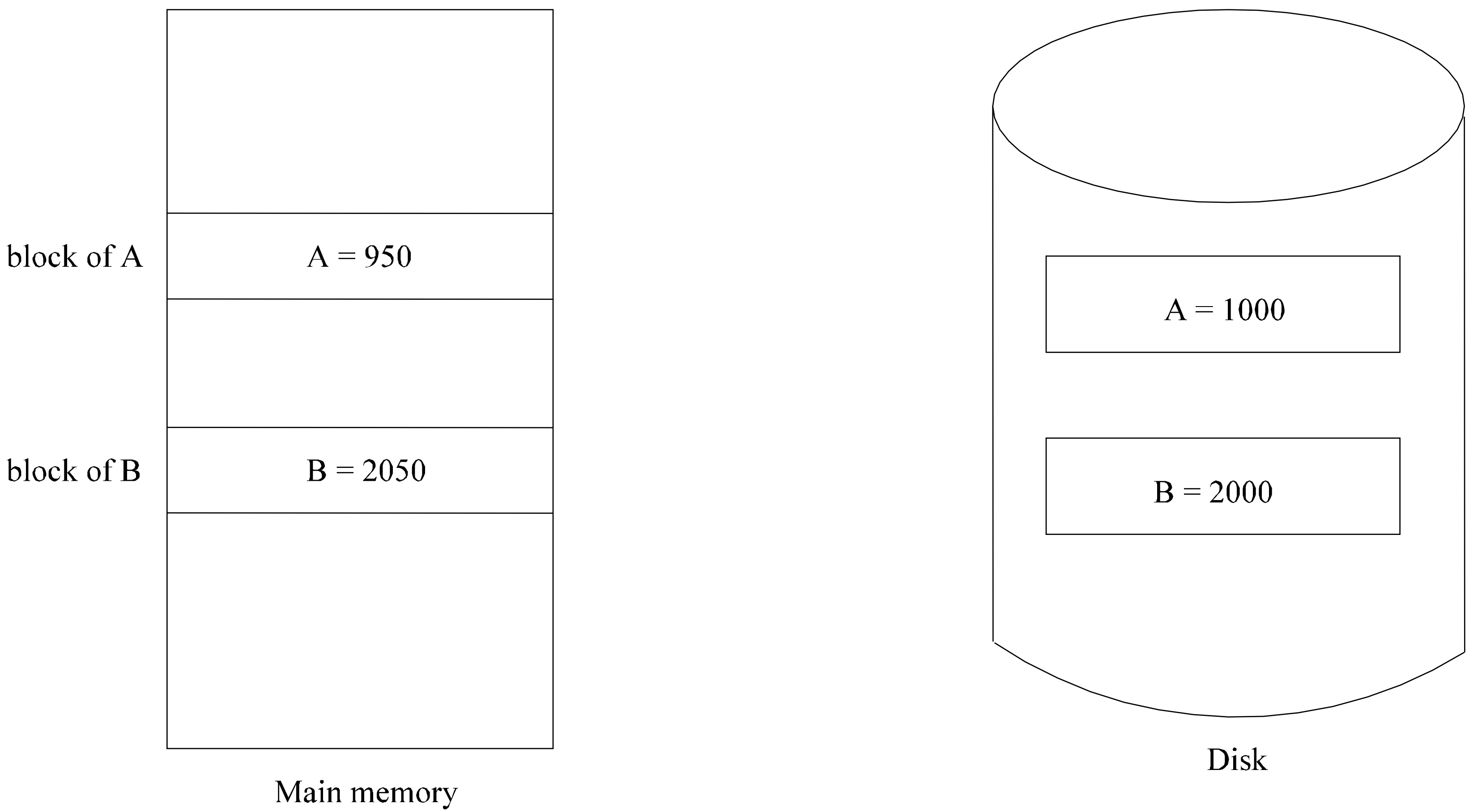
- read(A,a1): a1 ← A.
- write(A,a1)
- read(B,b1): input(B); b1 ← B

```
read(A,a1)
a1 := a1 - 50
write(A,a1)
read(B,b1)
b1 := b1 + 50
write(B,b1)
```



- Write(B,b1)

```
read(A,a1)
a1 := a1 - 50
write(A,a1)
read(B,b1)
b1 := b1 + 50
write(B,b1)
```



What is the state of the database after output(A), but before output(B)?
What if the system crashes at this moment?

Log-Based Recovery

- Recording database modifications in the *log*.
- Each record describes a single database write, and has the following fields:
  - Transaction name.
  - Data item name.
  - Old value (optional)
  - New value
- Special log record:
  - <Ti, start>
  - <Ti, commit>
  - <Ti, abort>

- *Require undo*: If it allows an uncommitted transaction to record in the stable database values it wrote.

- *Require redo*: If it allows a transaction to commit before all the values it wrote have been recorded in the stable database.

- *Undo Rule*: If x's location in the stable database presently contains the last committed value of x , then that value must be saved in stable storage (*log*) before being overwritten in the stable database by an uncommitted value.

- *Redo Rule*: Before a transaction can commit, the value it wrote for each data item must be in stable storage (*log*).

Immediate Database Modification (undo/redo)

- Allow database modifications to be output to the database while the transaction is active.

<u>T0:</u>
Read(A,a1)
a1 := a1 – 50
Write(A,a1)
Read(B,b1)
b1 := b1 + 50
Write(B,b1)

<u>T1:</u>
Read(C,c1)
c1 := c1 – 100
Write(C,c1)

<u>Log</u>:
<T0,starts>
<T0,A,1000,950>
<T0,B,2000,2050>
<T0,commits>
<T1,starts>
<T1,C,700,600>
<T1,commits>

Note: When T0 and T1 are executed concurrently, their log records may be interleaved.

- Transaction Ti needs to be undone if the log contains <Ti,starts> but does not contain <Ti,commits>.
- Transaction Ti needs to be redone if the log contains both <Ti, starts> and <Ti,commits>.

Log	Database
<T0,starts>	
<T0,A,1000,950>	
	A = 950
<T0,B,2000,2050>	
	B = 2050
<T0,commits>	
<T1,starts>	
<T1,C,700,600>	
	C = 600
<T1,commits>	

The write operations only update the data items in the buffer.
The updated values may not have been flushed to the hard disk.

<T0,starts>
<T0,A,1000,950>
<T0,B,2000,2050>

Undo T0

<T0,starts>
<T0,A,1000,950>
<T0,B,2000,2050>
<T0,commits>
<T1,starts>
<T1,C,700,600>

Redo T0, undo T1

<T0,starts>
<T0,A,1000,950>
<T0,B,2000,2050>
<T0,commits>
<T1,starts>
<T1,C,700,600>
<T1,commits>

Redo T0, redo T1

Note: Under concurrent execution, the log records from different transactions may interleave.

Detailed Procedure

- *Ti: Start*
  - Write <*Ti,start*> to log
- *Ti: Write(x,v)*
  - If x is not in the buffer, fetch it.
  - Append <*Ti,x,ov,v*> to the log.
  - Write *v* into the buffer slot occupied by *x*.
  - Acknowledge the scheduler.
- *Ti: Read(x)*
  - If *x* is not in the buffer, fetch it.
  - Return the value in *x*'s buffer slot to the scheduler.

- *Ti: Commit*
  - Write <*Ti, commit*>
  - Acknowledge the scheduler.
- *Ti: abort*
  - For each data item *x* updated by *Ti*
    - If *x* is not in the buffer, allocate a slot for it.
    - Copy the before image (*ov*) of *x* wrt *Ti* into *x's* buffer slot.
  - Write <*Ti, abort*>
  - Acknowledge the scheduler.

- *Restart*
  - Discard all buffer slots.
  - Let *redone* = {} and *undone* = {}.
  - Scan the log backward. Repeat the following steps until either $redone \cup undone$ equals the set of all data items, or there are no more log entries. For each log entry <*Ti,x,ov,v*>, if $x \notin redone \cup undone$, then
    - if *x* is not in the buffer, allocate a slot for it.
    - if the commit record of *Ti* has been found, copy *v* into *x*'s buffer slot and set $redone := redone \cup \{x\}$.
    - otherwise, copy the before image (*ov*) of *x* wrt *Ti* into *x's* buffer slot and set $undone := undone \cup \{x\}$.
  - Acknowledge the completion of Restart to the scheduler.

Example

<T0,starts>
<T0,A,1000,950>
<T1, starts>
<T1,C,700,600>
<T0,B,2000,2050>
<T0,commits>
<T1,A,950,1500>

Suppose after a crash the values of A, B and C found in the stable database are 1500, 2000 and 600 respectively.

Here is the log found in the stable storage.

From the log, we know that T0 has committed before the crash.

	Action (redo/undo)	A	B	C
		1500	2000	600
<T1,A,950,1500>	undo	950	2000	600
<T0,B,2000,2050>	redo	950	2050	600
<T1,C,700,600>	undo	950	2050	700
<T0,A,1000,950>	no action	950	2050	700

Deferred Database Modification (No-undo/redo)

- Recording all database modifications in the log, but deferring the execution of all write operations until the transaction commits.

Log	Database
<T0,starts>	
<T0,A,950>	
<T0,B,2050>	
<T0,commits>	
	A = 950
	B = 2050
<T1,starts>	
<T1,C,600>	
<T1,commits>	
	C = 600

Before a transaction commits, the update will not be written to the database (not even in the buffer).

After a transaction has committed, the data items in the buffer are updated. The updated values may not have been flushed to the hard disk.

<T0,starts>
<T0,A,950>
<T0,B,2050>

No action is needed

<T0,starts>
<T0,A,950>
<T0,B,2050>
<T0,commits>
<T1,starts>
<T1,C,600>

Redo T0

<T0,starts>
<T0,A,950>
<T0,B,2050>
<T0,commits>
<T1,starts>
<T1,C,600>
<T1,commits>

Redo T0, redo T1

Note: Under concurrent execution, the log records from different transactions may interleave.

Detailed Procedure

- *Ti: Start*
  - Write <*Ti,start*> to log
- *Ti: Write(x,v)*
  - Append <*Ti,x,v*> to the log.
  - Acknowledge the scheduler.
- *Ti: Read(x)*
  - If *Ti* has previously written into *x,* then return the after image of x wrt Ti.
  - Otherwise
    - If x is not in the buffer, fetch it.
    - Return the value in *x*'s buffer slot to the scheduler.

- *Ti: Commit*
  - Write <*Ti, commit*> to the log
  - For each *x* update by *Ti*
    - If *x* is not in the buffer, fetch it.
    - Copy the after image (*v*) of *x* wrt *Ti* into *x*'s buffer slot.
    - Acknowledge schedule
- *Ti: abort*
  - Write <*Ti, abort*>
  - Acknowledge the scheduler.

- *Restart*
  - Discard all buffer slots.
  - Let *redone* = {}.
  - Scan the log backward. Repeat the following steps until either *redone* equals the set of all data items, or there are no more log entries. For each log entry <*Ti,x,v*>, if the commit record of *Ti* has been found and $x \notin redone$, then
    - allocate a slot for *x* in the buffer;
    - Copy *v* into *x*'s buffer slot;
    - $redone := redone \cup \{x\}$.
  - Acknowledge the scheduler.

Example

<T0,starts>
<T0,A,950>
<T1, starts>
<T1,C,600>
<T0,B,2050>
<T0,commits>
<T1,A,1500>

Suppose after a crash the values of A, B and C found in the stable database are 950, 2000 and 700 respectively.

Here is the log found in the stable storage.

From the log, we know that T0 has committed before the crash.

	Action (redo)	A	B	C
		950	2000	700
<T1,A,1500>	No action	950	2000	700
<T0,B,2050>	redo	950	2050	700
<T1,C,600>	No action	950	2050	700
<T0,A,950>	redo	950	2050	700