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  - T_2 must abort because if it ever committed, the execution would no longer valid.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 use only those values that are written by committed transactions or by itself.

Strict Executions

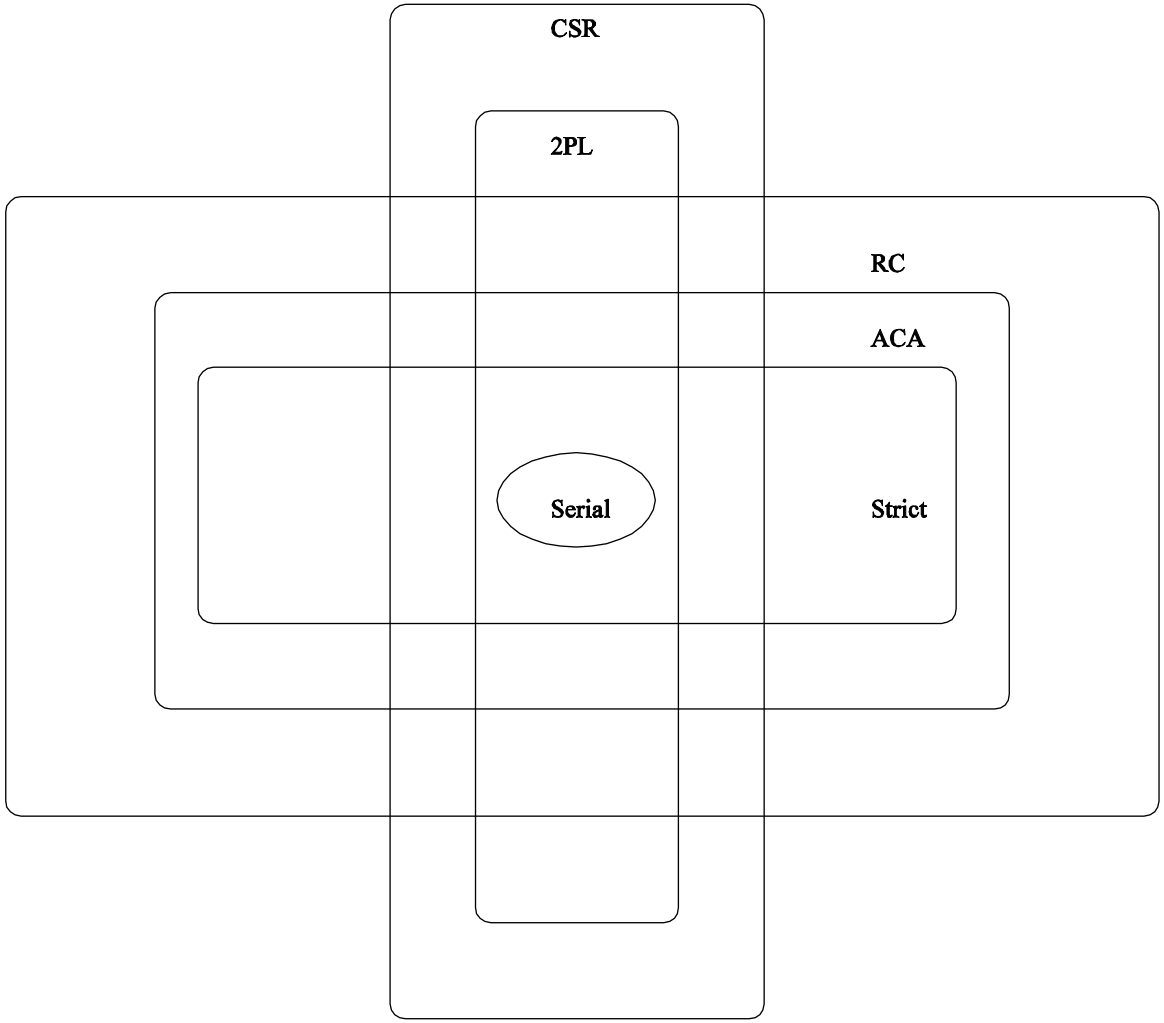
- Avoiding cascading aborts is not always enough from the practical point of view.
- $Write_1(x,1); Write_1(y,3); Write_2(y,1); Commit_1; read_2(x); Aborts_2$
- If we erase the operations from T_2, the resulting execution should be
  - $Write_1(x,1); Write_1(y,3); Commit_1$
- The value of *y* should 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.


CSR
2PL
RC
ACA
Serial
Strict

Transaction Execution

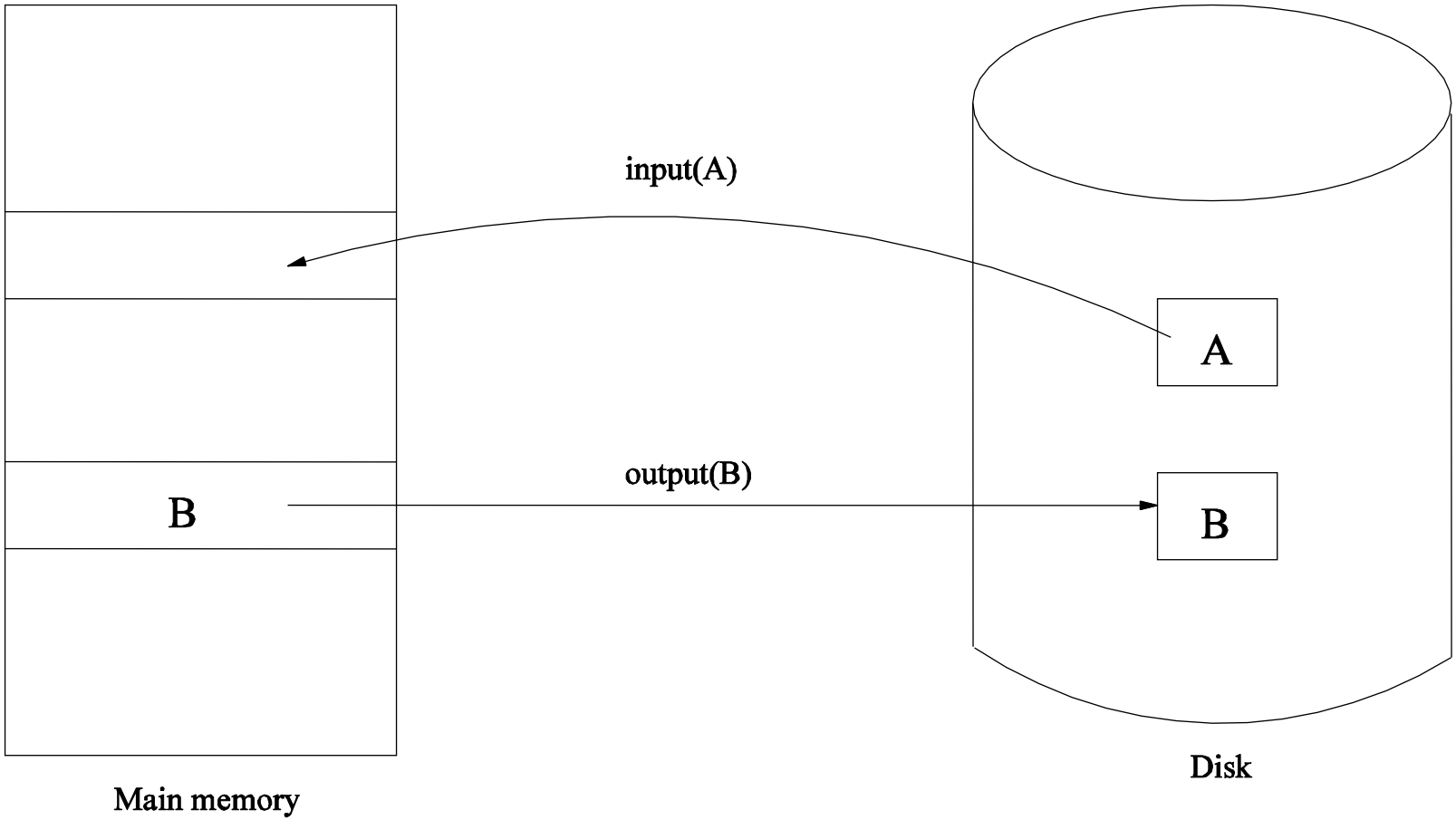
- Assume that strict two-phase locking protocol is used.
  - Locks are released at the end of a transaction.
  - If a transaction T_i writes a data item x, no transaction can read or write x before T_i commits or aborts.
  - When a transaction is aborted, the effect of a write operation can be removed 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

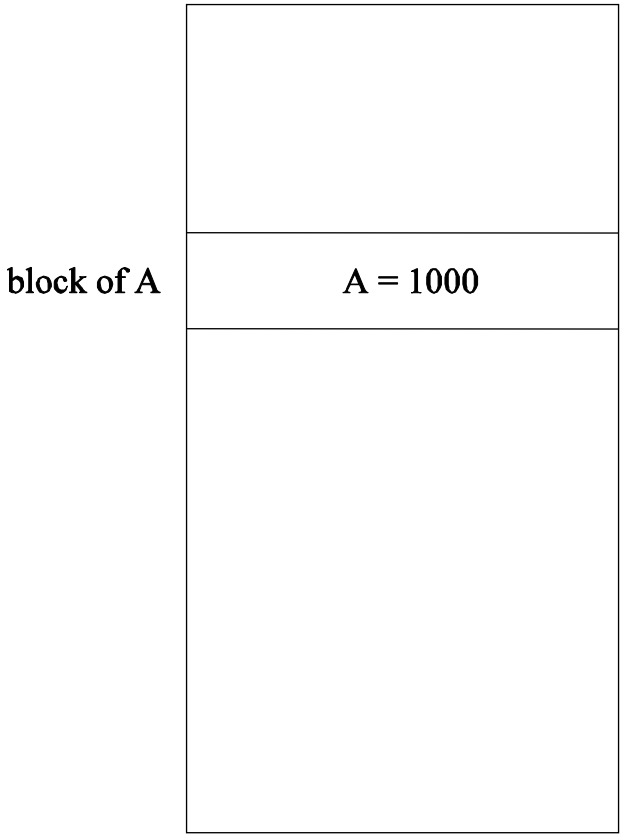
- Transactions interact with the database:
  - Data in database $\leftrightarrow$ program variables.
    - read(X,x_i): assigns the value of data item X to the local variable x_i.
      - Issue input(X), if X is not in the buffer.
      - Assign the value of X to x_i.
    - write(X,x_i): assigns the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

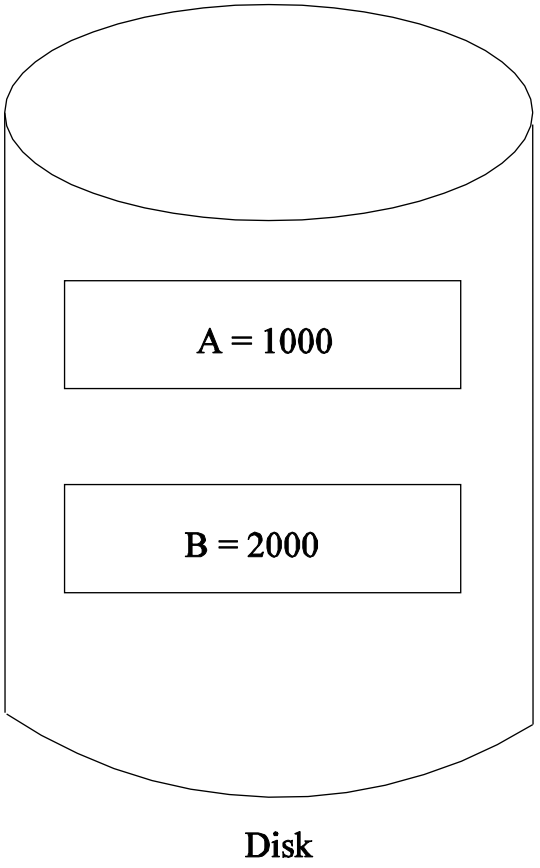
- Not specifically require the transfer of a block from buffer to disk.
- A buffer block is eventually written out to the disk either:
  - Buffer manager needs the memory space.
  - Force-output

Examples

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

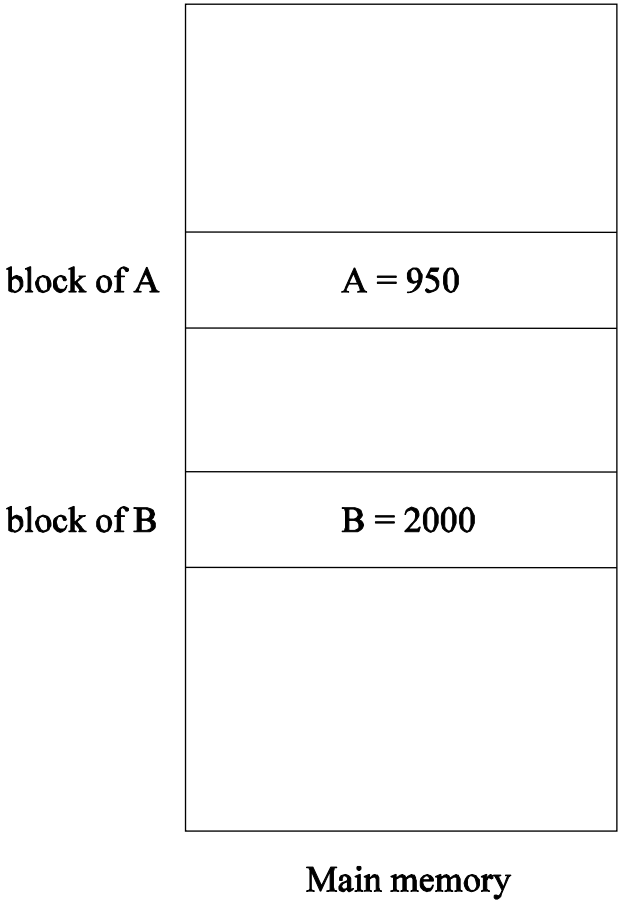
- The consistency constraint is that the sum of A and B is unchanged.
- Initial values of A and B are $1000 and $2000
- Main memory contains the buffer block A.

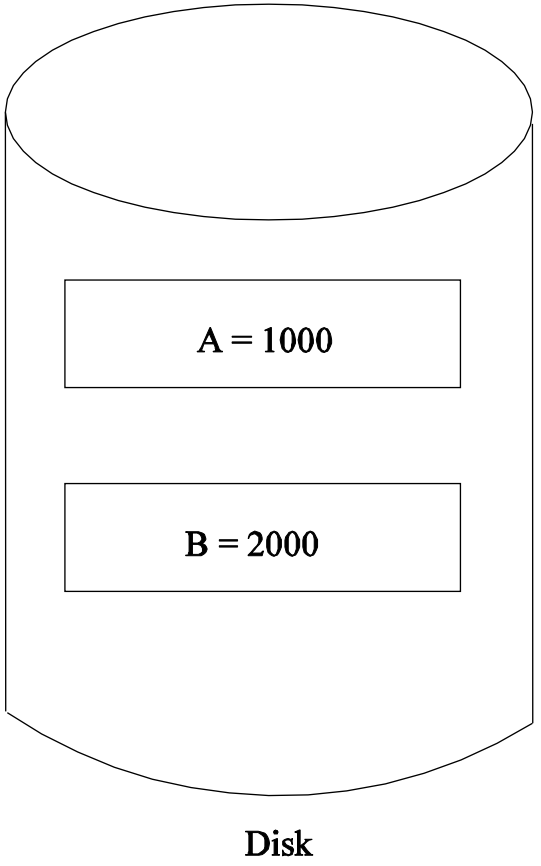

block of A
A = 1000
Main memory


A = 1000
B = 2000
Disk

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

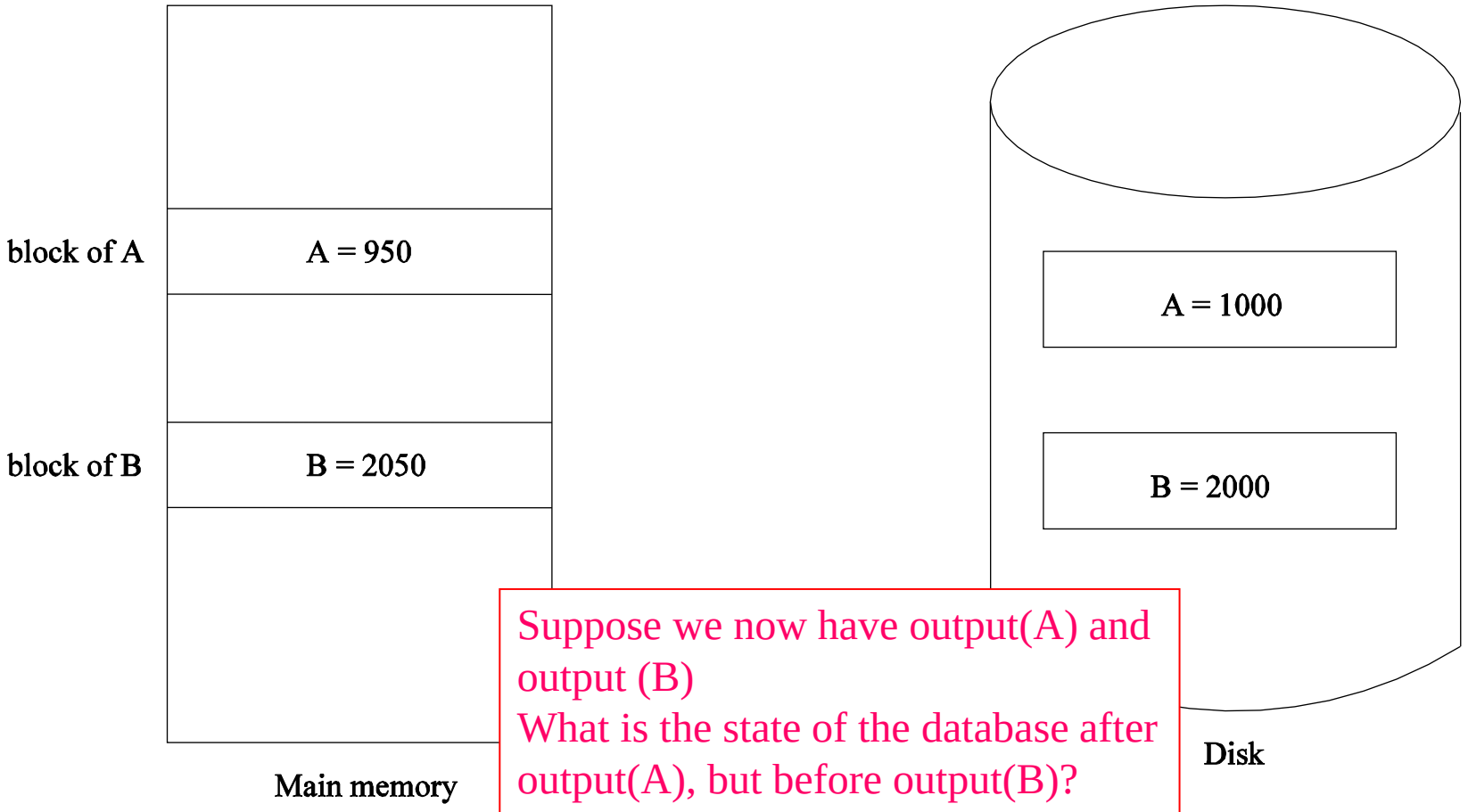
```
read(A,a1)
a1 := a1 - 50
write(A,a1)
read(B,b1)
b1 := b1 + 50
write(B,b1)
```





- Write(B,b1)

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



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

System Log

- To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items.
  1. The system needs to record the states information to recover failures correctly.
  2. The information is maintained in a log (also called journal or audit trail).
  3. The system log is kept in hard disk but maintains its current contents in main memory.

System Log

- 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

  E.g.(<T0,B,2000,2050>)

- Special log record:
  - <Ti, start>
  - <Ti, commit>
  - <Ti, abort>

System Log

Example

[start_transaction, T_1]
[read_item, T_1, A]
[read_item, T_1, D]
[write_item, T_1, D, 20, 25]
[commit, T_1]
[checkpoint]
[start_transaction, T_2]
[read_item, T_2, B]
[write_item, T_2, B, 12, 18]
[start_transaction, T_4]
[read_item, T_4, D]
[write_item, T_4, D, 25, 15]
[start_transaction, T_3]
[write_item, T_3, C, 30, 40]
[read_item, T_4, A]
[write_item, T_4, A, 30, 20]
[commit, T_4]
[read_item, T_2, D]
[write_item, T_2, D, 15, 25]

Immediate Update Techniques

- In the **immediate update** techniques, the database *may be updated* by some operations of a transaction *before* the transaction reaches its commit point.
- However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible.
- If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery.

Transaction Roll Back

- If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction. If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values

Undo, Redo with detail

- **Procedure** UNDO (WRITE_OP):
  - Undoing a write_item operation WRITE_OP
  - Consists of examining its log entry [write_item, *T*, *X*, old_value, new_value] and setting the value of item *X* in the database to old_value.
  - Undoing a number of write_item operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.
- **Procedure** REDO (WRITE_OP)**:**
  - Redoing a write_item operation WRITE_OP
  - Consists of examining its log entry [write_item, *T*, *X*, new_value] and setting the value of item *X* in the database to new_value.

Summary

- *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*).

Write-Ahead Logging (see later)

Stable Storage?

- "The *stable database* is organized as a set of pages that reside in stable storage. Here *stable storage* is an abstraction comprising all storage media that are resilient to the soft crash failures that we aim to recover from."
- "Most typically, this abstraction is implemented by secondary storage on magnetic disks"

https://www.sciencedirect.com/topics/computer-science/stable-storage

Database Recovery

- To maintain atomicity, a transaction's operations are redone or undone.
  - Undo: Restore all **BFIMs** on to disk.
  - Redo: Restore all **AFIMs** on to disk.
- Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two.
- These operations are recorded in the log as they happen.

- Note:
  - data values prior to modification (**BFIM** - BeFore Image)
  - new value after modification (**AFIM** – AFter Image)

Immediate Database Modification (undo/redo)

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

T0:
Read(A,a1)
a1 := a1 – 50
Write(A,a1)
Read(B,b1)
b1 := b1 + 50
Write(B,b1)

T1:
Read(C,c1)
c1 := c1 – 100
Write(C,c1)

Log:
<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.

NEW:

Context: The log is a sequence of log records, recording all the updated activities in the database

ADDED: Redoing committed transactions is necessary to ensure that all updates made during the transaction are **written to stable storage**. Even if a transaction is committed, there may still be unflushed pages in the buffer, and redoing helps bring the system to a consistent state after a failure

<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 **just before the crash**

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.

"It may be possible for a transaction T1 that all its log records have been output to **stable storage** but the actual updates on data are still in main memory. If a failure occurs at this point then redoing this transaction will ensure that all updates which were virtually lost due to failure would now get written to the stable storage."

<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 just **up to the crash**

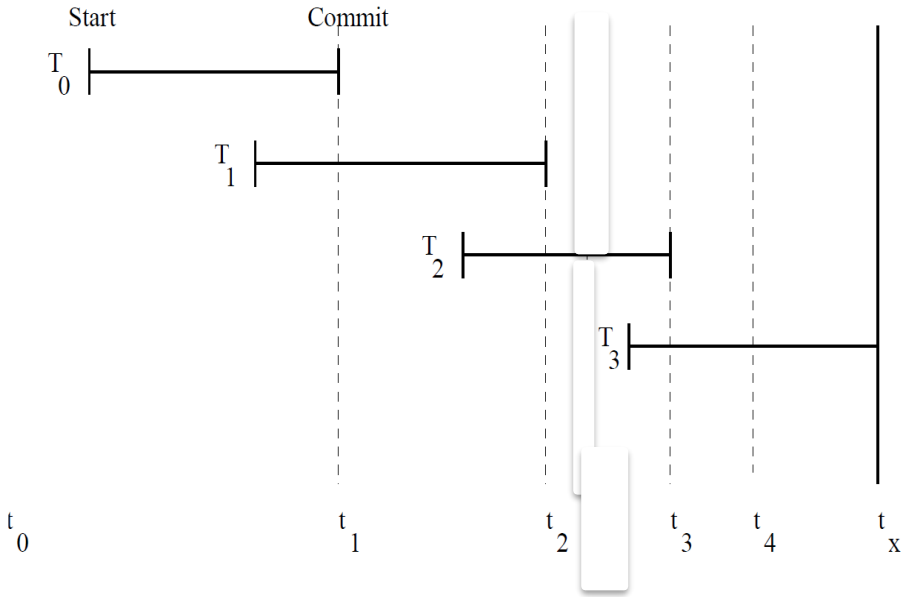
From the log, we know that T0 has committed before the crash.

	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

Summary

- Immediate Update:
  - As soon as a data item is modified in cache, the disk copy is updated (also applicable to any point before the transaction commits.)
- Deferred Update:
  - All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.

Log-based Recovery



- Assume that the database was recently created, the diagram shows transactions up until a crash.
- The database state will be somewhere between that at t_0 and the state at t_x (also true for log entries)

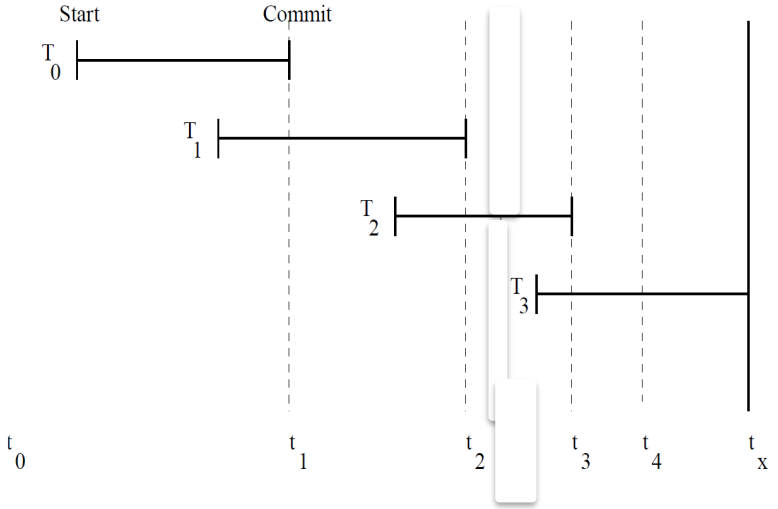
Write-Ahead Logging

- **Database reconvey is made possible by the Write-ahead log strategy**
  - "WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, i.e., after WAL records describing the changes have been flushed to permanent storage."
  - "If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the WAL records. (This is roll-forward recovery, also known as REDO.)"

Log Force-Write?

- Log file must be kept on disk
  - Only the log entries that have been written back to disk are considered in the recovery process.
- However, the most recent log exists in main memory
  - By now, you should grasp that before a transaction reaches its commit point, any portion of the **log** that has not been written to the disk yet must now be written to the disk.
  - This process is called **force-writing of the log file**, before committing a transaction*. A commit does not necessarily involve writing the data items to disk; this depends on the recovery mechanism in use.*
- When?
  - A commit is not necessarily required to initiate writing of the log file to disk. The log may sometimes be written back automatically when the **log buffe**r is full. This happens irregularly, as usually one block of the log file is kept in main memory until it is filled with log entries and then written back to disk, rather than writing it to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same information.

Log-based Recovery

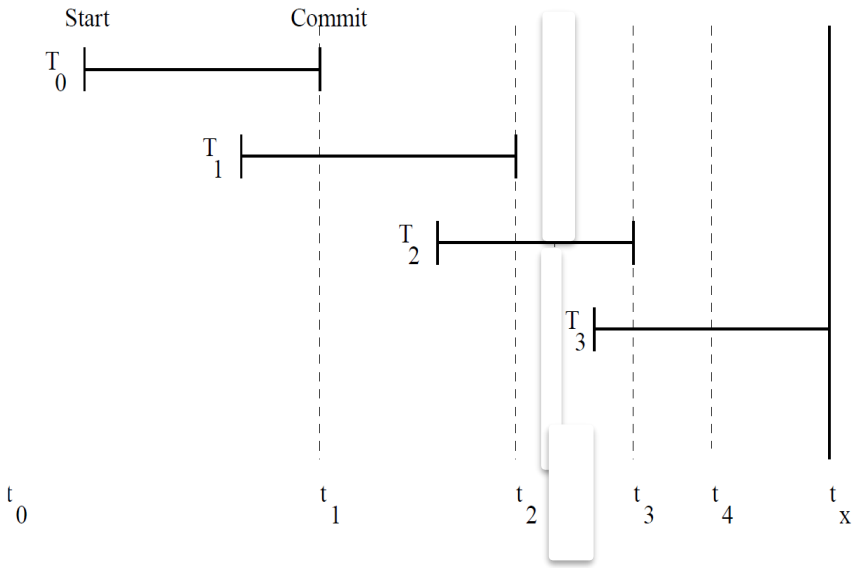


Recall: Only the log entries that have been written back to disk are considered in the recovery process.

- Suppose the log was last written to disk at t_4 (shortly after t_3)
- We would know that T_0, T_1 and T_2 have committed and their effects ~~**should be**~~ **need to be** reflected in the database after recovery.
- We also know that T_3 has started, may have modified some data, but is not committed. Thus T_3 should be undone.

Rolling back (Undo) T_3

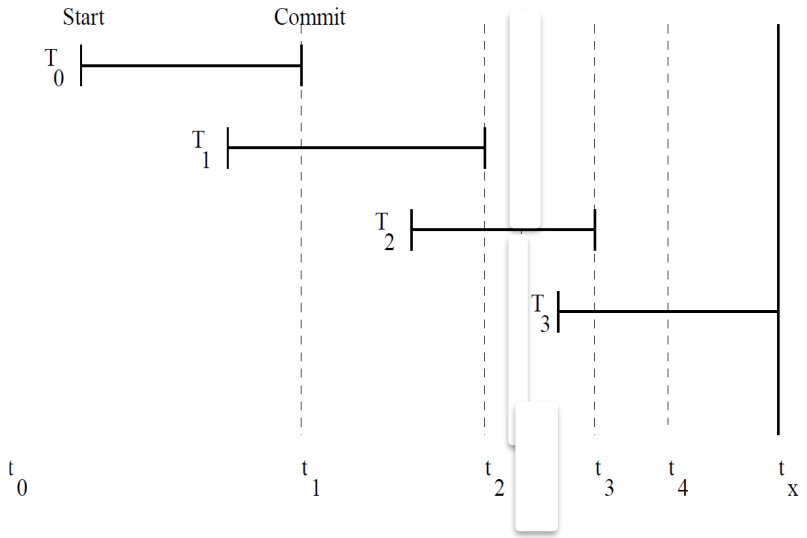
- With a write-ahead strategy, we would be able to make some recovery by rolling back T_3
- **Step 1:**
  - Undo the values written by T_3 to the old data values from the log



  - Undo helps guarantee **atomicity** in **ACID**

Redoing T_0 … T_2

- With a write-ahead strategy, we would be able to make some recovery by rolling back T_3
- **Step 2:**
  - Redoing the changes made by T_0 … T_2 using the new data values (for these committed transactions) from the log.

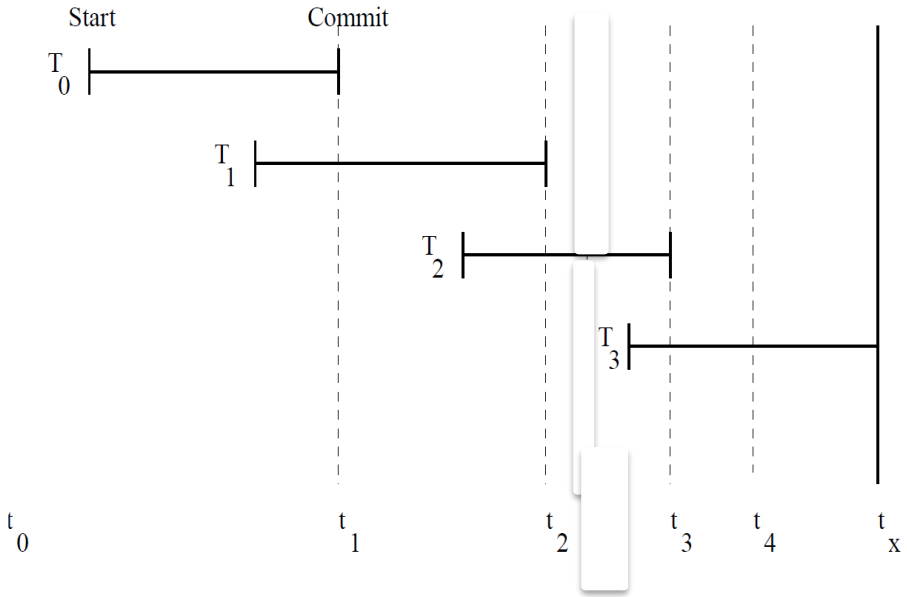


  - Redo guarantees **durability** in **ACID**

Checkpoints

- Notice also that using this system, the longer the time between crashes, the longer recovery may take.

Log-based Recovery Revisited



- Assume that the database was recently created, the diagram shows transactions up until a crash.
  - Note this assumption!
  - **Recall again**: Only the log entries that have been written back to disk are considered in the recovery process.
- The database state will be somewhere between that at t_0 and the state at t_x (also true for log entries)

NEW

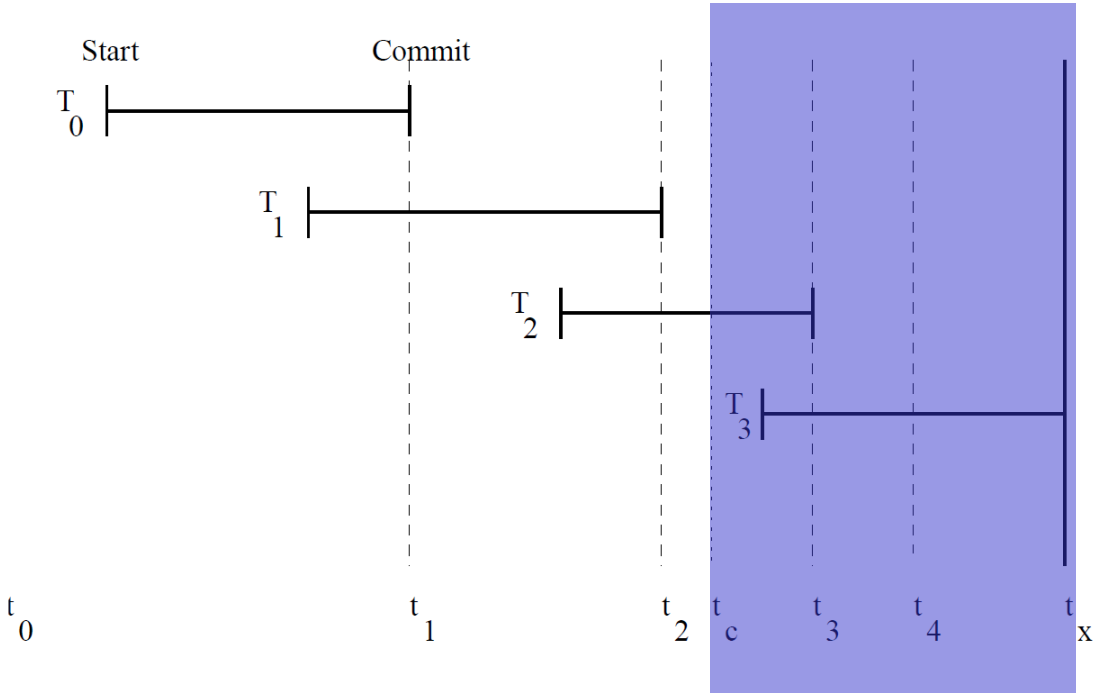
- The problem for the recovery manager is to determine which operations need to be considered and which can safely be ignored. This problem is usually overcome by writing another kind of entry in the log: the checkpoint entry.
- The **checkpoint** is written into the log periodically and always involves the writing out to the database on disk the effect of all write operations of committed transactions. Hence, all transactions that have their commit(T) entries in the log before a checkpoint entry will not require their write operations to be redone in case of a system crash.

https://www.cs.uct.ac.za/mit_notes/database/htmls/chp14.html#committing-transactions-and-force-writing

Checkpoints

- To reduce this problem, the system could take *checkpoints* at regular intervals.
- Taking a checkpoint consists of the following actions:
  1. Suspend execution of transactions temporarily.
  2. Force-write **ALL** main memory buffers that have been modified to disk.
  3. Write a [checkpoint] record to the log, and force-write the log to disk.
  4. Resume executing transactions.

Log Checkpoints



- In our example, suppose a checkpoint is taken at time t_c. Then on recovery we only need redo T_2.

Recall Failures

1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution.
2. **A transaction or system error.** Some operation in the transaction may cause it to fail.
3. **Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction
4. **Concurrency control enforcement.** The concurrency control method may decide to abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions.

Catastrophic Failures

1. **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
2. **Physical problems and catastrophes.** This refers to an endless list of problems that includes power or fire, theft, overwriting disks by mistake…

Catastrophic Failures

- So far, all the techniques we have discussed apply to non-catastrophic failures
- A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure.
- The recovery techniques we have discussed use the entries in the system log to recover from failure by bringing the database back to a consistent state.
- The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes.

Database Backup

- The main technique used to handle such crashes is a **database backup**
  - (1) whole database and (2) the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices.
  - The latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.
- To not lose all transactions they have performed since the last database backup. We also back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape.