Unit 11: Recovery System

Recovery

- An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.
- The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a failure.

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition such as bad input, data not found.
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- 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

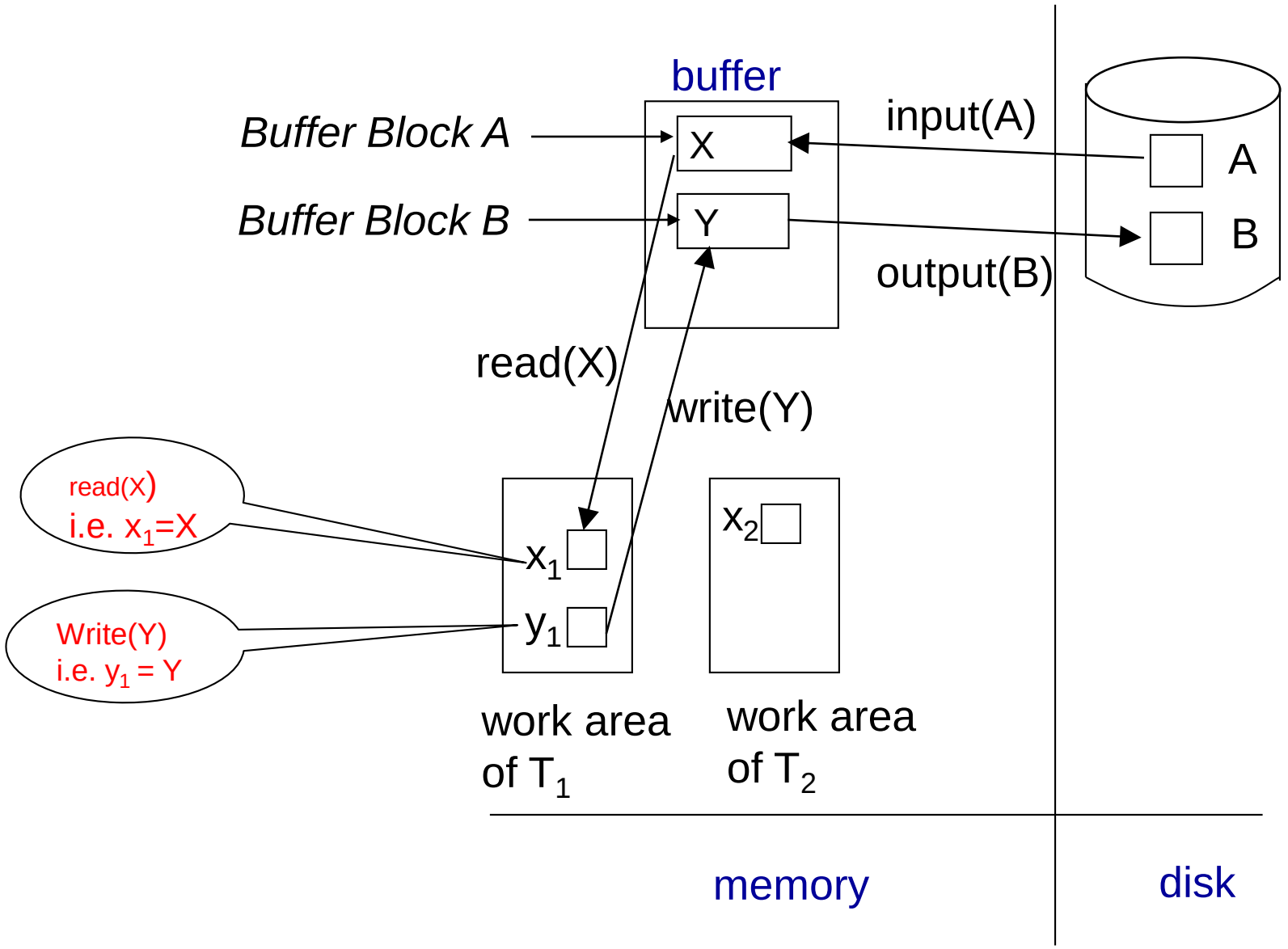
Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
    - Example: RAID systems.

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

Example of Data Access



Data Access (Cont.)

- 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.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **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.
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

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 <T_i **start**>log record
- An update log record describes a single database write
- *Before* T_i executes **write**(X), a log record
  <T_i, X, V_1, V_2>
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.

Log-Based Recovery(Cont.)

Two approaches using logs

- Deferred database modification
- Immediate database modification

- A transaction **creates a log record** prior to modifying the database.
- The log records **allow** the system to **undo changes** made by a transaction in the event that the transaction must be aborted;
- Also **allow** the system also to **redo changes** made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk.

Log-Based Recovery(Cont.)

Consider the **steps a transaction takes in modifying a data item**:

1. The transaction performs some computations in its own work area of main memory.

2. The transaction modifies the data block in the disk buffer in main memory holding the data item **write(x)**.

3. The database system executes the **output(x)** operation that writes the data block to disk.

If before output(x) system crashes

Deferred database modification

- The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits.

- A transaction is said to be **partially committed** once the final action of the transaction has been executed.

- The deferred-modification technique described here section **assumes that transactions are executed serially.**

- When a transaction partially commits, **the information on the log associated with the transaction is used in executing the deferred writes.**

- If the **system crashes before the transaction completes its execution**, **or** if the transaction **aborts**, then the information on the **log is simply ignored**.

Deferred database modification

To illustrate, reconsider our simplified banking system.

Suppose that **these transactions are executed serially**, in the order T_0 followed by T_1.

The values of accounts **A, B**, and **C** before the execution took place were **$1000, $2000, and $700**, respectively

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).

The execution of transaction *Ti* proceeds as follows.

- Before *Ti* starts its execution, a record **<*Ti* start>** is written to the log.
- A write(*X*) operation by *Ti* results in the writing of a new record to the log. **<Ti, A, old-value, new-value>**
- Finally, when *Ti* partially commits, a record **<*Ti* commit>** is written to the log.

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

Log	Database
$<T_0$ start$>$	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0$ commit$>$	
$<T_1$ start$>$	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1$ commit$>$	

Deferred database modification

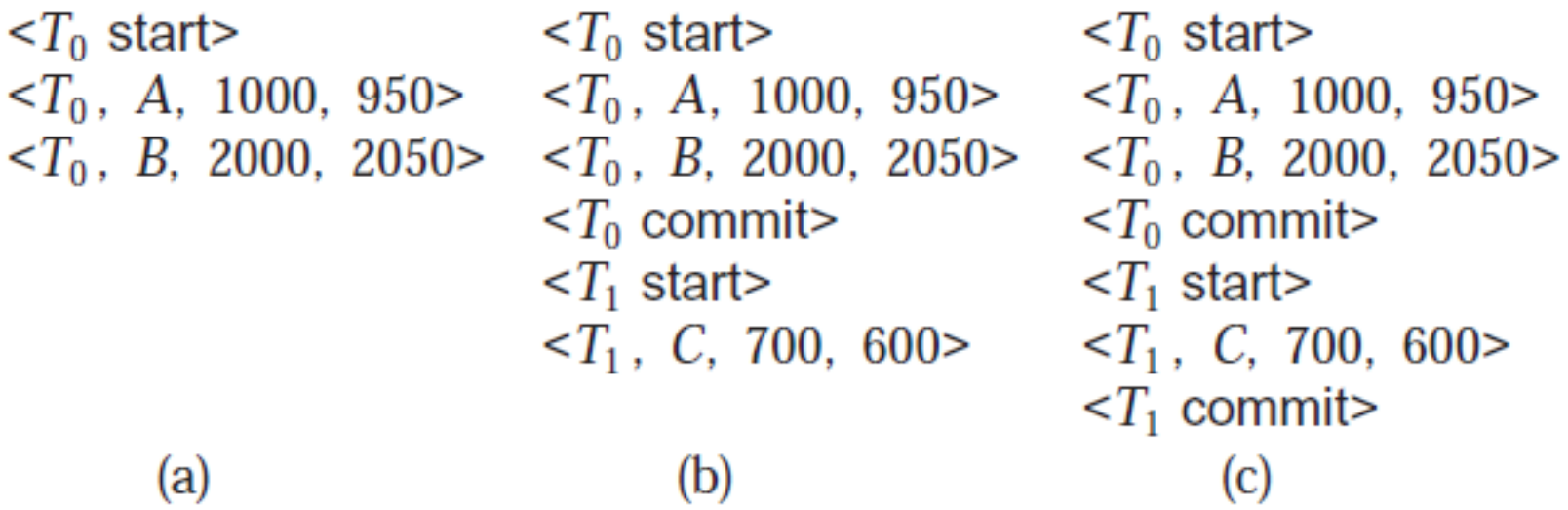
- When transaction T_i partially commits, the records associated with it in the log are used in executing the deferred writes.

- Since a failure may occur while this updating is taking place, we must **ensure that**, before the start of these updates, all the **log records are written out to stable storage**.
- Once they have been written, the **actual updating takes place**, and the transaction enters the committed state.

  - Transaction T_i needs to be **redone if** and only if the log contains both the record **<Ti start>** and the record **<Ti commit>.**

    - Thus, if the **system crashes after the transaction completes** its execution
      - Recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.
    - Otherwise
      - Log is ignored

Deferred DB Modification Recovery Example



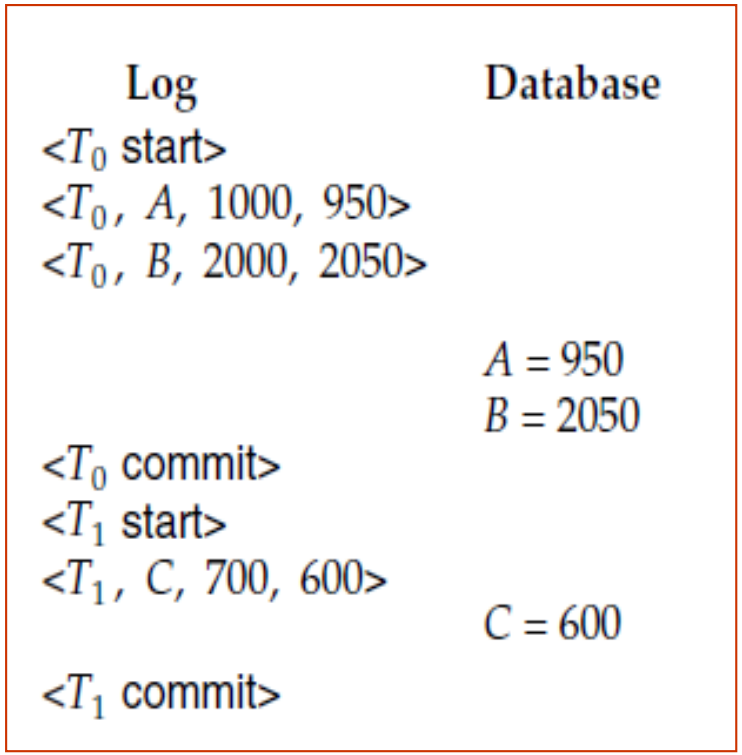
(a) NO- redo(T0): When the system comes back up, **no redo actions** need to be taken, since **no commit** record appears in the log.

(b) redo(T0) :When the system comes back up, the operation **redo(T0)** is **performed**, since the record <T0 commit> appears in the log on the disk.

(c) redo(T0) and redo(T1):When the system comes back up, two commit records are in the log: one for T0 and one for T1. Therefore, the system **must perform** operations **redo(T0) and redo(T1), in the order** in which their commit records appear in the log.

Immediate Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
- Output of updated blocks to stable storage 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.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy



Undo and Redo Operations-Immediate Database Modification

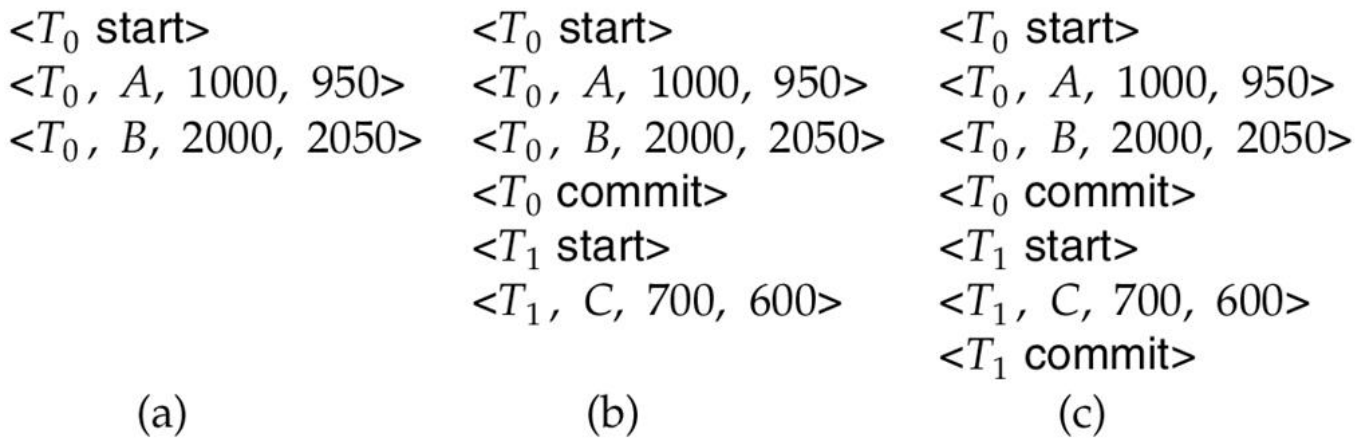
- **Undo** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out
    - when undo of a transaction is complete, a log record <T_i **abort**> is written out.
  - **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
    - No logging is done in this case

Undo and Redo on Recovering from Failure
Immediate Database Modification

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**>

Immediate DB Modification Recovery Example

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



Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

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

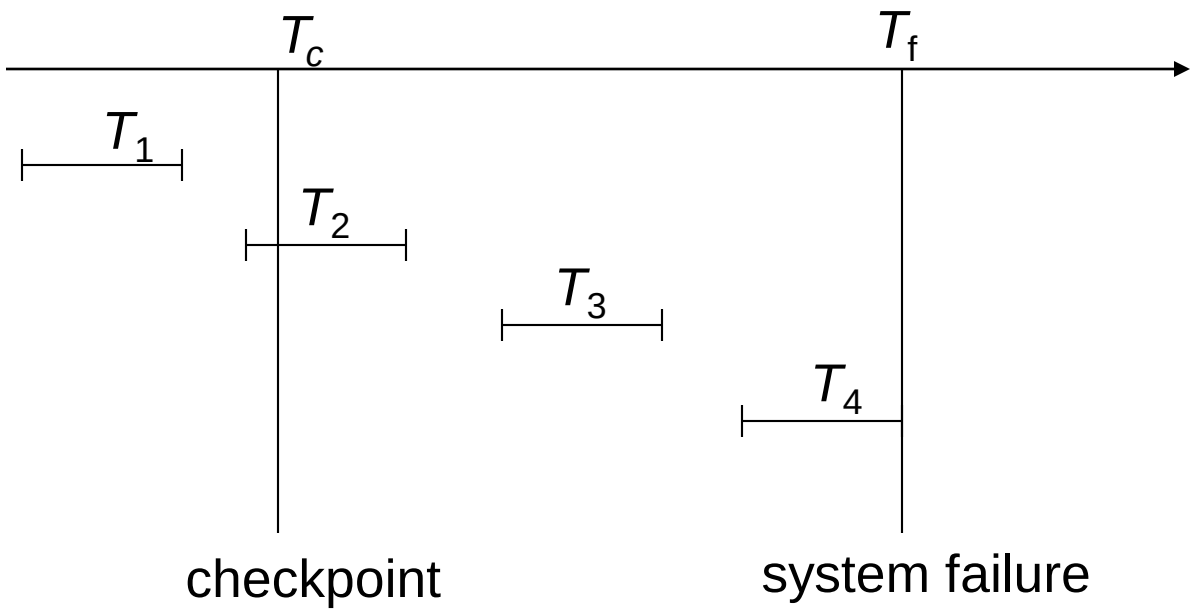
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
     - All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  1. Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone