# Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **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
  - See book for more details on how to implement stable storage

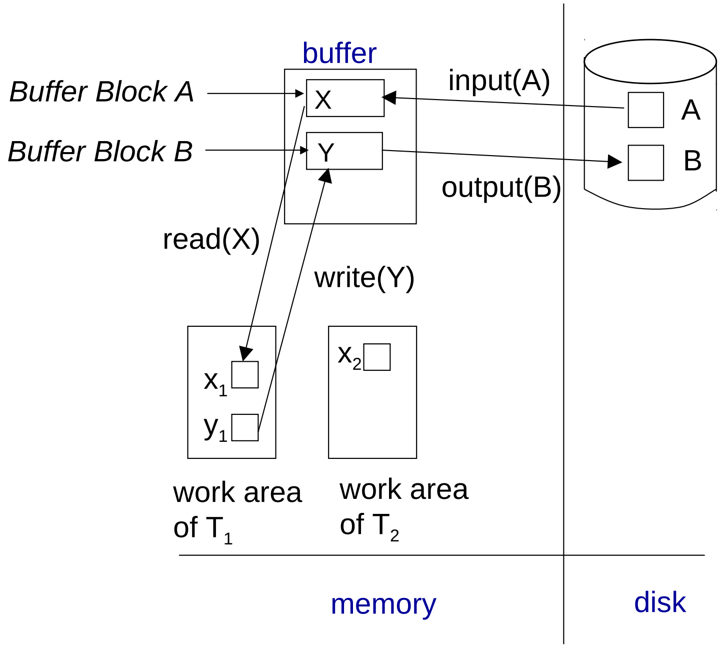
# Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

# 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

# Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-paging** (brief details in book)

# 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
- *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 <$T_i$ **commit**> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

# Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - all previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

# 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

# Immediate Database Modification Example

| **Log** | **Write** | **Output** |
|---|---|---|
| <$T_0$ **start**> | | |
| <$T_0$, A, 1000, 950> | | |
| <$T_o$, B, 2000, 2050 | | |
| | $A$ = 950<br>$B$ = 2050 | |
| <$T_0$ **commit**> | | |
| <$T_1$ **start**> | | |
| <$T_1$, C, 700, 600> | | |
| | $C$ = 600 | |
| | | $B_B$, $B_C$ — $B_C$ output before $T_1$ commits |
| <$T_1$ **commit**> | | |
| | | $B_A$ — $B_A$ output after $T_0$ commits |

# Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction $T_i$ has modified an item, no other transaction can modify the same item until $T_i$ has committed or aborted*
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

# Undo and Redo Operations

- **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

- When recovering after failure:
  - Transaction $T_i$ needs to be undone if the log
    - contains the record <$T_i$ **start**>,
    - but does not contain either the record <$T_i$ **commit**> *or* <$T_i$ **abort**>.
  - Transaction $T_i$ needs to be redone if the log
    - contains the records <$T_i$ **start**>
    - and contains the record <$T_i$ **commit**> *or* <$T_i$ **abort**>
- Note that If transaction $T_i$ was undone earlier and the <$T_i$ **abort**> record written to the log, and then a failure occurs, on recovery from failure $T_i$ is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

# Immediate DB Modification Recovery Example

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

| (a) | (b) | (c) |
|---|---|---|
| <$T_0$ start> | <$T_0$ start> | <$T_0$ start> |
| <$T_0$, $A$, 1000, 950> | <$T_0$, $A$, 1000, 950> | <$T_0$, $A$, 1000, 950> |
| <$T_0$, $B$, 2000, 2050> | <$T_0$, $B$, 2000, 2050> | <$T_0$, $B$, 2000, 2050> |
| | <$T_0$ commit> | <$T_0$ commit> |
| | <$T_1$ start> | <$T_1$ start> |
| | <$T_1$, $C$, 700, 600> | <$T_1$, $C$, 700, 600> |
| | | <$T_1$ commit> |

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

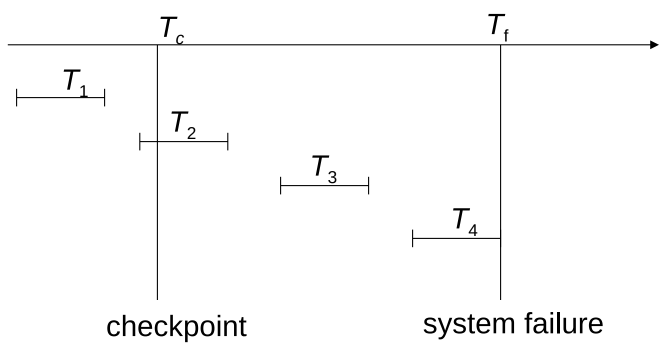
## Checkpoints

- 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

## Recovery Algorithm

- **So far:** we covered key concepts
- **Now**: we present the components of the basic recovery algorithm
- **Later**: we present extensions to allow more concurrency

# Recovery Algorithm

- **Logging** (during normal operation):
  - <$T_i$ **start**> at transaction start
  - <$T_i$, $X_j$, $V_1$, $V_2$> for each update, and
  - <$T_i$ **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - Let $T_i$ be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of $T_i$ of the form <$T_i$, $X_j$, $V_1$, $V_2$>
    - perform the undo by writing $V_1$ to $X_j$,
    - write a log record <$T_i$ , $X_j$, $V_1$>
      - such log records are called **compensation log records**
  - Once the record <$T_i$ **start**> is found stop the scan and write the log record <$T_i$ **abort**>

# Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last <**checkpoint** $L$> record, and set undo-list to $L$.
  2. Scan forward from above <**checkpoint** $L$> record
     1. Whenever a record <$T_i$, $X_j$, $V_1$, $V_2$> is found, redo it by writing $V_2$ to $X_j$
     2. Whenever a log record <$T_i$ **start**> is found, add $T_i$ to undo-list
     3. Whenever a log record <$T_i$ **commit**> *or* <$T_i$ **abort**> is found, remove $T_i$ from undo-list

# Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <$T_i$, $X_j$, $V_1$, $V_2$> is found where $T_i$ is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing $V_1$ to $X_j$.
        2. write a log record <$T_i$ , $X_j$, $V_1$>
     2. Whenever a log record <$T_i$ **start**> is found where $T_i$ is in undo-list,
        1. Write a log record <$T_i$ **abort**>
        2. Remove $T_i$ from undo-list
     3. Stop when undo-list is empty
        - i.e. <$T_i$ **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

# Example of Recovery