# Recovery System

**These slides are a modified version of the slides of the book "Database System Concepts" (Chapter 17), 5th Ed., McGraw-Hill, by Silberschatz, Korth and Sudarshan.**
**Original slides are available at www.db-book.com**

# Transactions: ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

*Consistency: Programmer*

*Isolation: Concurrency Control System*

*Atomicity and Durability: Recovery System*

# Transaction

- commit
  termination with success of the transaction
  *all operations are executed and changes to the database are persistent*
- abort (or rollback)
  abort of the transaction
  *none operation is executed*

Transfers $50 from account *A* to account *B*

```
start transaction;
update Account
     set balance = balance + $50 where Account_number = B;
update Account
     set balance = balance – $50  where Accout_number = A;
commit;
```

# Transaction

1) Abort if balance of A less than $50

```
start transaction;
update Accont
        set balance = balance + $50 where Account_number = B;
update Account
        set balance = balance – $50  where Accout_number = A;
select balance into V
        from Account where Account_number = A;
if (V>=0)    then commit
             else abort;
```

2) Abort if the system has entered an undesirable state (e.g. deadlock)

# Failures

- A computer system is subject to failures
- Causes are: disk failure, power outage, hardware or software errors, ….
- In any failure, information may be lost
- DBMS must take actions in advance to ensure that atomicity and durability properties of transactions are preserved
- **Recovery System**:
  it can restore the database to the consistent state that existed before the failure

ASSUMPTIONS:

- **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

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

- 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

So far we saw different storage media in databases (RAID Levels)

Resilience to failure classification:

- **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
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - **Information residing in stable storage is never lost!!!**
    (theoretically never cannot be guaranteed - it can be closely approximated by techniques that make data loss extremely unlikely)

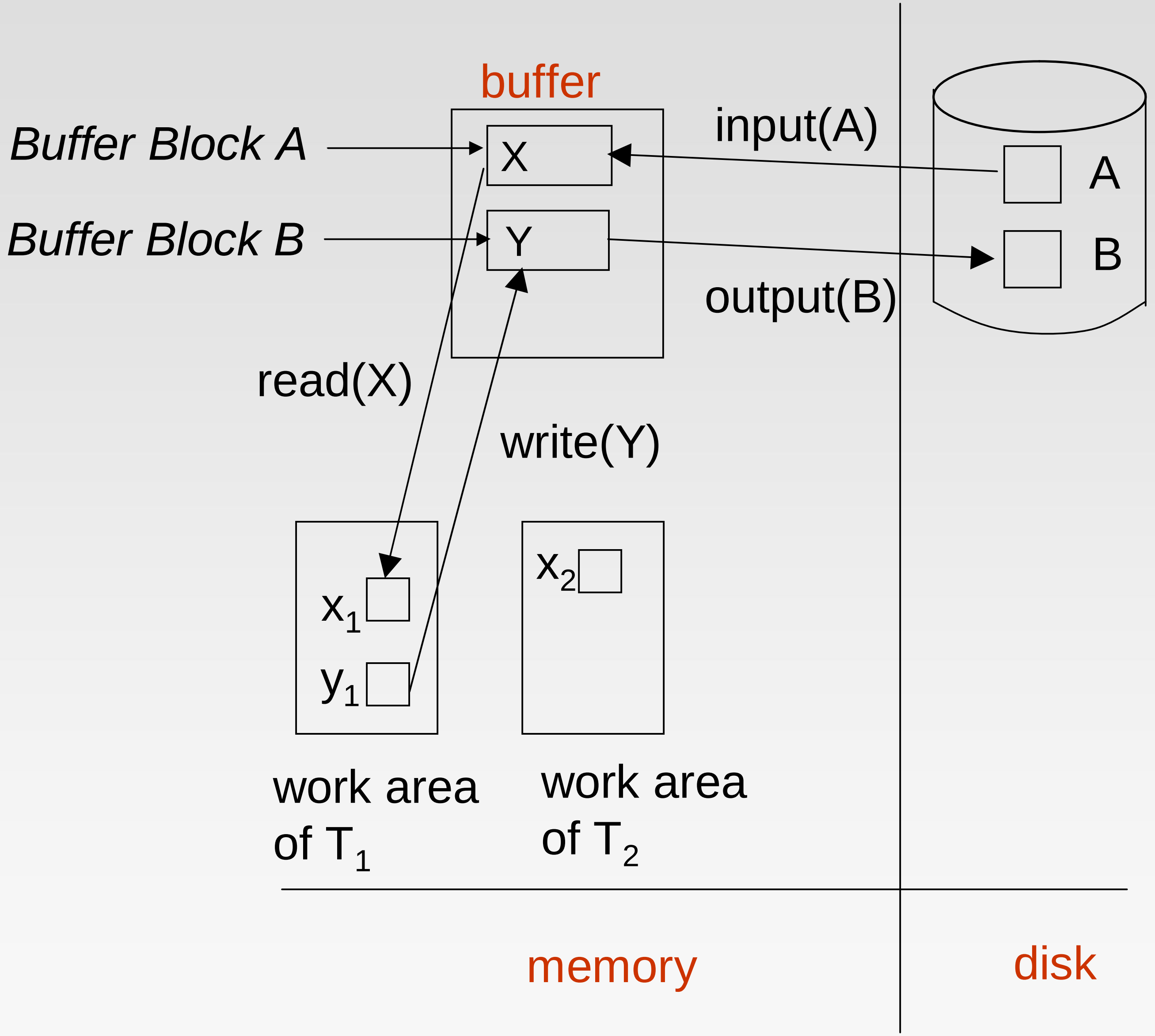
# Data Access

- **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
  - **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 log record

  <$T_i$ **start**>
- *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, 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 it last statement, the log record

  <$T_i$ **commit**>

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

*insert(X): log record <$T_i$, X, empty, V2>*

*delete(X): log record <$T_i$, X, V1, empty>*

# Log-Based Recovery

- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

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

- When $T_i$ partially commits, <$T_i$ **commit**> is written to the log

# Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Write(X):
  - old value of X is not needed in the log file for this scheme
- Transaction starts by writing <$T_i$ ***start***> record to log.
- A **write**(*X*) operation results in a log record <$T_i$, *X, V*> being written, where *V* is the new value for *X*
- The write is not performed on *X* at this time, but is deferred.
- When $T_i$ partially commits, <$T_i$ **commit**> 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 <$T_i$ **start**> and<$T_i$ **commit**> 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.

| | | |
|---|---|---|
| <$T_0$ start> | <$T_0$ start> | <$T_0$ start> |
| <$T_0$, $A$, 950> | <$T_0$, $A$, 950> | <$T_0$, $A$, 950> |
| <$T_0$, $B$, 2050> | <$T_0$, $B$, 2050> | <$T_0$, $B$, 2050> |
| | <$T_0$ commit> | <$T_0$ commit> |
| | <$T_1$ start> | <$T_1$ start> |
| | <$T_1$, $C$, 600> | <$T_1$, $C$, 600> |
| | | <$T_1$ commit> |
| (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 <$T_0$ **commi**t> is present
  - (c) **redo**($T_0$) must be performed followed by redo($T_1$) since <$T_0$ **commit**> and <$T_i$ **commit**> are present

# Immediate Database Modification

- The **immediate database modification** scheme allows database updates (**output**(*…*)) 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> | | |
| <$T_o$, B, 2000, 2050> | | |
| | $A$ = 950 | |
| | $B$ = 2050 | |
| <$T_0$ **commit**> | | |
| <$T_1$ **start**> | | |
| <$T_1$, C, 700, 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 <$T_i$ **start**>, but does not contain the record <$T_i$ **commit**>.
  - Transaction $T_i$ needs to be redone if the log contains both the record <$T_i$ **start**> and the record <$T_i$ **commit**>.
- Undo operations are performed first, then redo operations.

# 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.

(b) undo ($T_1$) and redo ($T_0$): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(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

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.

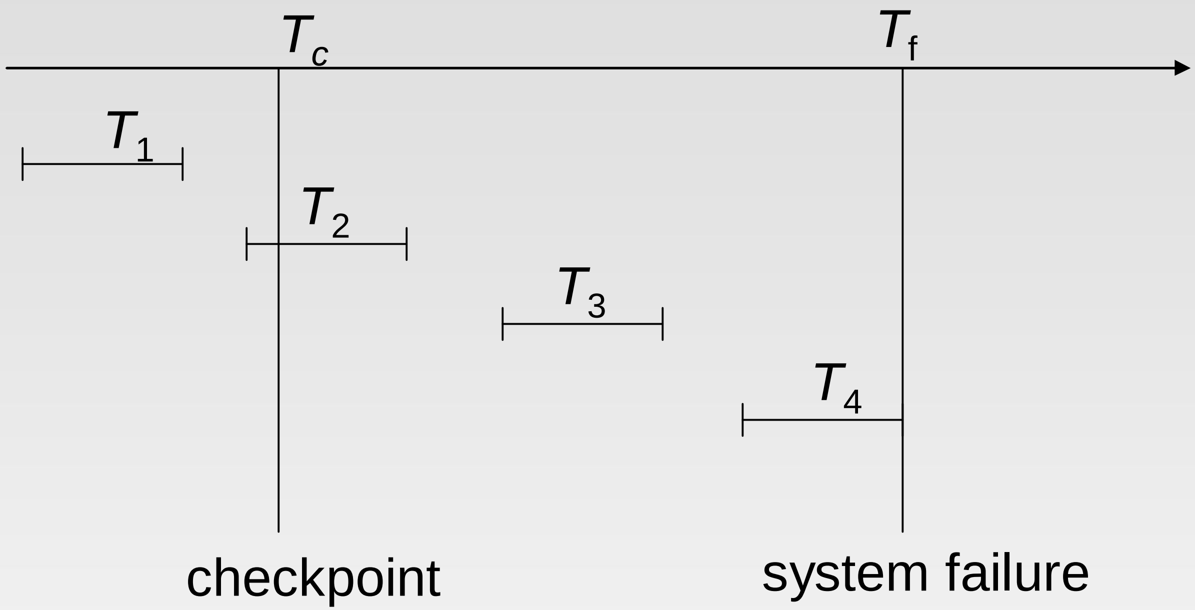
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all **log records** currently residing in main memory onto stable storage.
  2. **Output all modified buffer blocks to the disk**.
  3. Write a log record < **checkpoint**> onto stable storage

Transactions are not allowed to execute any actions while a checkpoint is in progress.

# 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**> record
  2. Continue scanning backwards till a record <$T_i$ **start**> is found for transaction in the checkpoint.
  3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from $T_i$ or later) with no <$T_i$ **commit**>, execute **undo**(*$T_i$*). (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from $T_i$ or later with a <$T_i$ **commit**>, execute **redo**(*$T_i$*).

# 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 With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - 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 concurrency control using **strict two-phase locking**;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
- Logging is done as described earlier.
  - Log records of different transactions may be interspersed in the log.
- The checkpointing technique and actions taken on recovery have to be changed
  - since several transactions may be active when a checkpoint is performed.

# Recovery With Concurrent Transactions (Cont.)

- Checkpoints are performed as before, except that the checkpoint log record is now of the form

  < **checkpoint** *L*>

where *L* is the list of **transactions active** at the time of the checkpoint

- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.
     For each record found during the backward scan:
     - if the record is <$T_i$ **commit**>, add $T_i$ to *redo-list*
     - if the record is <$T_i$ **start**>, then if $T_i$ is not in *redo-list*, add $T_i$ to *undo-list*
  3. For every $T_i$ in *L*, if $T_i$ is not in *redo-list*, add $T_i$ to *undo-list*

# Recovery With Concurrent Transactions (Cont.)

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when <$T_i$ **start**> records have been encountered for every $T_i$ in *L*.
     - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  2. Scan log forwards from the <$T_i$ **start**> oldest record found at step 1 till the end of the log.
     - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

# Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

  <$T_0$ **star**t>
  <$T_0$, *A*, 0, 10>
  <$T_0$ **commit**>
  <$T_1$ **start**>  /* Scan at step 1 comes up to here */
  <$T_1$, *B*, 0, 10>
  <$T_2$ **start**>
  <$T_2$, *C*, 0, 10>
  <$T_2$, *C*, 10, 20>
  <checkpoint {$T_1$, $T_2$}>
  <$T_3$ **start**>
  <$T_3$, *A*, 10, 20>
  <$T_3$, *D*, 0, 10>
  <$T_3$ **commit**>

# Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.

Several log records can thus be output using a single output operation, reducing the I/O cost.

# Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction $T_i$ enters the commit state only when the log record <$T_i$ **commit**> has been output to stable storage. **Log force** is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

# Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk.

# Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

# Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
  - Apply the Log Recovery