Chapter 13: Recovery System
Chapter 13: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Recovery With Concurrent Transactions
- Failure with Loss of Nonvolatile Storage
- Remote Backup Systems
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

- 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

- **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
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.
- 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

Buffer Block A
Buffer Block B

read(X)
write(Y)

input(A)
output(B)

work area of T1
work area of T2

memory
disk
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.

  - 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 `<$T_i$ 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, and $V_2$ is the value to be written to $X$.
  - Log record notes that $T_i$ has performed a write on data item $X$. $X_j$ had value $V_1$ before the write, and will have value $V_2$ after the write.
- When $T_i$ finishes its 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).
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification
Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing \(<T_i \ text{ start}>\) 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 \(X\)
  - Note: old value is not needed for this scheme
- The write is not performed on \(X\) at this time, but is deferred.
- When \(T_i\) partially commits, \(<T_i \ text{ commit}>\) 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 $<T_i \text{ start}>$ and $<T_i \text{ commit}>$ 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 $(C)$
Portion of the Database Log Corresponding to $T_0$ and $T_1$

\[
<T_0 \text{ start}> \\
<T_0, A, 950> \\
<T_0, B, 2050> \\
<T_0 \text{ commit}> \\
<T_1 \text{ start}> \\
<T_1, C, 600> \\
<T_1 \text{ commit}>
\]
State of the Log and Database Corresponding to $T_0$ and $T_1$

<table>
<thead>
<tr>
<th>Log</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;T_0 \text{ start}&gt;$</td>
<td>$A = 950$</td>
</tr>
<tr>
<td>$&lt;T_0, A, 950&gt;$</td>
<td>$B = 2050$</td>
</tr>
<tr>
<td>$&lt;T_0, B, 2050&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;T_0 \text{ commit}&gt;$</td>
<td>$C = 600$</td>
</tr>
<tr>
<td>$&lt;T_1 \text{ start}&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1, C, 600&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1 \text{ commit}&gt;$</td>
<td></td>
</tr>
</tbody>
</table>
Deferred Database Modification (Cont.)

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

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;T_0 \text{ start}&gt;$</td>
<td>$&lt;T_0 \text{ start}&gt;$</td>
<td>$&lt;T_0 \text{ start}&gt;$</td>
</tr>
<tr>
<td>$&lt;T_0, A, 950&gt;$</td>
<td>$&lt;T_0, A, 950&gt;$</td>
<td>$&lt;T_0, A, 950&gt;$</td>
</tr>
<tr>
<td>$&lt;T_0, B, 2050&gt;$</td>
<td>$&lt;T_0, B, 2050&gt;$</td>
<td>$&lt;T_0, B, 2050&gt;$</td>
</tr>
<tr>
<td>$&lt;T_0 \text{ commit}&gt;$</td>
<td>$&lt;T_0 \text{ commit}&gt;$</td>
<td>$&lt;T_0 \text{ commit}&gt;$</td>
</tr>
<tr>
<td>$&lt;T_1 \text{ start}&gt;$</td>
<td>$&lt;T_1 \text{ start}&gt;$</td>
<td>$&lt;T_1 \text{ start}&gt;$</td>
</tr>
<tr>
<td>$&lt;T_1, C, 600&gt;$</td>
<td>$&lt;T_1, C, 600&gt;$</td>
<td>$&lt;T_1 \text{ commit}&gt;$</td>
</tr>
</tbody>
</table>

- 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 $<T_0 \text{ commit}>$ is present
  (c) redo($T_0$) must be performed followed by redo($T_1$) since $<T_0 \text{ commit}>$ and $<T_i \text{ commit}>$ are present
Immediate Database Modification

- The **immediate database modification** scheme allows database updates 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

<table>
<thead>
<tr>
<th>Log</th>
<th>Write</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;T_0 \text{ start}&gt;$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$&lt;T_0, A, 1000, 950&gt;$</td>
<td></td>
<td>$A = 950$</td>
</tr>
<tr>
<td>$&lt;T_0, B, 2000, 2050&gt;$</td>
<td></td>
<td>$B = 2050$</td>
</tr>
<tr>
<td>$&lt;T_0 \text{ commit}&gt;$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1 \text{ start}&gt;$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1, C, 700, 600&gt;$</td>
<td></td>
<td>$C = 600$</td>
</tr>
<tr>
<td>$&lt;T_1 \text{ commit}&gt;$</td>
<td></td>
<td>$B_B, B_C$</td>
</tr>
</tbody>
</table>

- Note: $B_X$ denotes block containing $X$. 
Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - $\text{undo}(T_i)$ restores the value of all data items updated by $T_i$ to their old values, going backwards from the last log record for $T_i$
  - $\text{redo}(T_i)$ sets the value of all data items updated by $T_i$ to the new values, going forward from the first log record for $T_i$

- When recovering after failure:
  - Transaction $T_i$ needs to be undone if the log contains the record $<T_i \text{ start}>$, but does not contain the record $<T_i \text{ commit}>$.
  - Transaction $T_i$ needs to be redone if the log contains both the record $<T_i \text{ start}>$ and the record $<T_i \text{ commit}>$.

- Undo operations are performed first, thenredo operations.
Immediate DB Modification Recovery Example

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

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;T_0 \text{ start}&gt;)</td>
<td>(&lt;T_0 \text{ start}&gt;)</td>
<td>(&lt;T_0 \text{ start}&gt;)</td>
</tr>
<tr>
<td>(&lt;T_0, A, 1000, 950&gt;)</td>
<td>(&lt;T_0, A, 1000, 950&gt;)</td>
<td>(&lt;T_0, A, 1000, 950&gt;)</td>
</tr>
<tr>
<td>(&lt;T_0, B, 2000, 2050&gt;)</td>
<td>(&lt;T_0, B, 2000, 2050&gt;)</td>
<td>(&lt;T_0, B, 2000, 2050&gt;)</td>
</tr>
<tr>
<td>(&lt;T_0 \text{ commit}&gt;)</td>
<td>(&lt;T_0 \text{ commit}&gt;)</td>
<td>(&lt;T_0 \text{ commit}&gt;)</td>
</tr>
<tr>
<td>(&lt;T_1 \text{ start}&gt;)</td>
<td>(&lt;T_1 \text{ start}&gt;)</td>
<td>(&lt;T_1 \text{ start}&gt;)</td>
</tr>
<tr>
<td>(&lt;T_1, C, 700, 600&gt;)</td>
<td>(&lt;T_1, C, 700, 600&gt;)</td>
<td>(&lt;T_1, C, 700, 600&gt;)</td>
</tr>
</tbody>
</table>

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.
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.
  
  3. Need only consider the part of log following above `start` 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
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?

- 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
  \(<\text{checkpoint } L>\)
  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 \(\text{undo-list}\) and \(\text{redo-list}\) to empty
  2. Scan the log backwards from the end, stopping when the first \(<\text{checkpoint } L>\) record is found.
     For each record found during the backward scan:
     - if the record is \(<T_i \text{ commit}>\), add \(T_i\) to \(\text{redo-list}\)
     - if the record is \(<T_i \text{ start}>\), then if \(T_i\) is not in \(\text{redo-list}\), add \(T_i\) to \(\text{undo-list}\)
  3. For every \(T_i\) in \(L\), if \(T_i\) is not in \(\text{redo-list}\), add \(T_i\) to \(\text{undo-list}\)
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 \(<T_i \text{ start}>\) records have been encountered for every \(T_i\) in *undo-list*.
   - During the scan, perform *undo* for each log record that belongs to a transaction in *undo-list*.
2. Locate the most recent \(<\text{checkpoint } L>\) record.
3. Scan log forwards from the \(<\text{checkpoint } L>\) record till the end of the log.
   - During the scan, perform *redo* for each log record that belongs to a transaction on *redo-list*
Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead 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.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- 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 $<T_i \text{commit}>$ has been output 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
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 \(<\text{dump}\)> 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
- Can be extended to allow transactions to be active during dump; known as fuzzy dump or online dump
Remote Backup Systems
Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.
Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages

- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the log records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.
Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.

- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.

- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.

- **One-safe:** commit as soon as transaction’s commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.

- **Two-very-safe:** commit when transaction’s commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.

- **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as its commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.
End of Chapter 13