Recovery
**Review: The ACID properties**

**Atomicity**: All actions in the Xaction happen, or none happen.

**Consistency**: If each Xaction is consistent, and the DB starts consistent, it ends up consistent.

**Isolation**: Execution of one Xaction is isolated from that of other Xacts.

**Durability**: If a Xaction commits, its effects persist.

CC guarantees **Isolation and Consistency**.

The **Recovery Manager** guarantees **Atomicity & Durability**.
Why is recovery system necessary?

- **Transaction failure**: application errors (e.g., div by 0, segmentation fault)
  - Logical errors: application errors (e.g., div by 0, segmentation fault)
  - System errors: deadlocks

- **System crash**: hardware/software failure causes the system to crash.

- **Disk failure**: head crash or similar disk failure destroys all or part of disk storage

Lost data can be in main memory or on disk
Storage Media

n **Volatile storage:**
  • does not survive system crashes
  • examples: main memory, cache memory

n **Nonvolatile storage:**
  • survives system crashes
  • examples: disk, tape, flash memory, non-volatile (battery backed up) RAM

n **Stable storage:**
  • a “mythical” form of storage that survives all failures
  • approximated by maintaining multiple copies on distinct nonvolatile media
Recovery and Durability

To achieve Durability: Put data on stable storage

To approximate stable storage make two copies of data

Problem: data transfer failure
Recovery and Atomicity

- Durability is achieved by making 2 copies of data

- What about atomicity…
  
  - Crash may cause inconsistencies…
Recovery and Atomicity

Example: transfer $50 from account A to account B
• goal is either to perform all database modifications made by \( Ti \) or none at all.

Requires several inputs (reads) and outputs (writes)

Failure after output to account A and before output to B….  
• DB is corrupted!
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 and durability.
**Background: Data Access**

- **Physical blocks**: blocks on disk.
- **Buffer blocks**: blocks in main memory.

**Data transfer:**
- **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 $x_i$.

Assumption: 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**(BX) instruction before the assignment, if the block BX 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**(BX) need not immediately follow **write**(X).
➢ System can perform the **output** operation when it deems fit.
Buffer Block A

Buffer Block B

read(X)

write(Y)

input(A)

output(B)

work area of T1

work area of T2

memor y

disk
To ensure atomicity, first output information about modifications to stable storage without modifying the database itself.

We study two approaches:

- log-based recovery, and
- shadow-paging
Log-Based Recovery

Simplifying assumptions:
• Transactions run serially
• logs are written directly on the stable storage

Log: a sequence of log records; maintains a record of update activities on the database. (Write Ahead Log, W.A.L.)

Log records for transaction Ti:
• \(<Ti \text{ start } >\)
• \(<Ti, X, V1, V2 >\)
• \(<Ti \text{ commit } >\)

Two approaches using logs
• Deferred database modification
• Immediate database modification
### Log example

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>&lt;T1, start&gt;</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>&lt;T1, A, 1000, 950&gt;</td>
</tr>
<tr>
<td>Write(A)</td>
<td>&lt;T1, B, 2000, 2050&gt;</td>
</tr>
<tr>
<td>Read(B)</td>
<td>&lt;T1, commit&gt;</td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>
Deferred Database Modification

- Ti starts: write a \(<Ti\ start>\) record to log.

- Ti \textbf{write}(X)
  - write \(<Ti, X, V>\) to log: \textit{V is the new value for X}
  - The write is deferred

- ➢ Note: old value is \textbf{not} needed for this scheme

- Ti partially commits:
  - Write \(<Ti\ commit>\) to the log

- DB updates by reading and executing the log:
  - \(<Ti\ start>\) ...... \(<Ti\ commit>\)
Deferred Database Modification

How to use the log for recovery after a crash?

Redo: if both `<Ti start>` and `<Ti commit>` are there in the log.

Crashes can occur while

- the transaction is executing the original updates, or
- while recovery action is being taken

Example transactions `T0` and `T1` (`T0` executes before `T1`):

- `T0`: read (A)
  - `A`: - A - 50
  - write (A)
- `T1`: read (C)
  - `C`: - C - 100
  - write (C)
- read (B)
  - `B`: - B + 50
- write (B)
Below we show the log as it appears at three instances of time.

(a) <T0, start> <T0, A, 950> <T0, B, 2050> <T0, commit> <T1, start> <T1, C, 600>

(b) <T1, commit>

What is the correct recovery action in each case?
Immediate Database Modification

- Database updates of an uncommitted transaction are allowed

- Tighter logging rules are needed to ensure transactions are undoable
  - LOG records must be of the form: \(<Ti, X, V_{old}, V_{new}>\)
  - Log record must be written *before* database item is written
  - Output of DB blocks can occur:
    - Before or after commit
    - In any order
Immediate Database Modification (Cont.)

Recovery procedure:

- **Undo**: `<Ti, start>` is in the log but `<Ti commit>` is not. Undo:
  - restore the value of all data items updated by `Ti` to their old values, going backwards from the last log record for `Ti`
- **Redo**: `<Ti start>` and `<Ti commit>` are both in the log.
  - sets the value of all data items updated by `Ti` to the new values, going forward from the first log record for `Ti`

Both operations must be **idempotent**: even if the operation is executed multiple times the effect is the same as if it is executed once
### Immediate Database Modification Example

<table>
<thead>
<tr>
<th>Log</th>
<th>Write</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;T0 start&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;T0, A, 1000, 950&gt;</code></td>
<td>A = 950</td>
<td></td>
</tr>
<tr>
<td><code>&lt;T0, B, 2000, 2050&gt;</code></td>
<td>B = 2050</td>
<td></td>
</tr>
<tr>
<td><code>&lt;T0 commit&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;T1 start&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;T1, C, 700, 600&gt;</code></td>
<td>C = 600</td>
<td>BB, BC</td>
</tr>
<tr>
<td><code>&lt;T1 commit&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: BX denotes block containing X.
Recovery actions in each case above are:
(a) **undo** (*T0*): B is restored to 2000 and A to 1000.
(b) **undo** (*T1*) and **redo** (*T0*): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
(c) **redo** (*T0*) and **redo** (*T1*): A and B are set to 950 and 2050 respectively. Then C is set to 600
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.

How to avoid redundant redoes?

- Put marks in the log indicating that at that point DB and log are consistent. Checkpoint!
Checkpoints

At a checkpoint:

- Quiesce system operation.

- Output all log records currently residing in main memory onto stable storage.

- Output all modified buffer blocks to the disk.

- Write a log record `<checkpoint>` onto stable storage.
Recovering from log with checkpoints:

1. Scan backwards from end of log to find the most recent `<checkpoint>` record

2. Continue scanning backwards till a record `<Ti start>` is found.

3. Need only consider the part of log following above start record. Why?

4. After that, recover from log with the rules that we had before.
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
Shadow Paging

- **Shadow paging**: alternative to log-based recovery; works mainly for serial execution of transactions

- Keeps “clean” data (the shadow pages) untouched during transaction (in stable storage)

- Writes to a copy of the data

- Replace the shadow page only when the transaction is committed and output to the disk
Shadow Paging

- Maintain two page tables during the lifetime of a transaction – the current page table, and the shadow page table.
- Store the shadow page table in nonvolatile storage,
  - Shadow page table is never modified during execution.
- To start with, both page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time,
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy.
  - The update is performed on the copy.
Sample Page Table

page table

pages on disk
Example of Shadow Paging
Shadow and current page tables after write to page 4
Shadow Paging

To commit a transaction:

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page table the new shadow page table, as follows:
   ➢ keep a pointer to the shadow page table at a fixed (known) location on disk.
   ➢ to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
• Once pointer to shadow page table has been written, transaction is committed.

No recovery is needed after a crash! — new transactions can start right away, using the shadow page table.
Shadow Paging

Advantages

• no overhead of writing log records
• recovery is trivial

Disadvantages:

• Copying the entire page table is very expensive
• Data gets fragmented
• Hard to extend for concurrent transactions
Recovery With Concurrent Transactions

To permit concurrency:

- All transactions share a single disk buffer and a single log
- Concurrency control: Strict 2PL: i.e. Release eXclusive locks only after commit.
- Logging is done as described earlier.

The checkpointing technique and actions taken on recovery have to be changed (based on ARIES)

since several transactions may be active when a checkpoint is performed.
Checkpoints for concurrent transactions:

\(<\text{checkpoint } L>\)

\(L:\) the list of transactions active at the time of the checkpoint

- We assume no updates are in progress while the checkpoint is carried out

Recovery for concurrent transactions, 3 phases:

1. Initialize \(undo-list\) and \(redo-list\) 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:
     1. if the record is \(<T_i \text{commit}>\), add \(T_i\) to \(redo-list\)
     2. if the record is \(<T_i \text{start}>\), 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

---

n Scan log backwards
• Perform undo(\(T\)) for every transaction in undo-list
• Stop when you have seen \(<T, \text{ start}>\) for every \(T\) in undo-list.

n Locate the most recent \(<\text{checkpoint } L>\) record.
1. Scan log forwards from the \(<\text{checkpoint } L>\) record till the end of the log.
è perform redo for each log record that belongs to a transaction on redo-list

---

UNDO

---

REDO
Example of Recovery

:  

\(<T_0 \text{ start}>\)
\(<T_0, A, 0, 10>\)
\(<T_0 \text{ commit}>\)
\(<T_1 \text{ start}>\)
\(<T_1, B, 0, 10>\)
\(<T_2 \text{ start}>\)
\(<T_2, C, 0, 10>\)
\(<T_2, C, 10, 20>\)
\(<\text{checkpoint \{T}_1, T_2\}>\)
\(<T_3 \text{ start}>\)
\(<T_3, A, 10, 20>\)
\(<T_3, D, 0, 10>\)
\(<T_3 \text{ commit}>\)

Redo-list\{T}_3\}
Undo-list\{T}_1, T_2\}

\textbf{Undo:}
Set C to 10
Set C to 0
Set B to 0

\textbf{Redo:}
Set A to 20
Set D to 10

\begin{tabular}{|c|c|c|c|c|}
\hline
DB & A  & B  & C  & D  \\
\hline
Initial & 0  & 0  & 0  & 0  \\
\hline
At crash & 20 & 10 & 20 & 10 \\
\hline
After rec. & 20 & 0  & 0  & 10 \\
\hline
\end{tabular}
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 performs 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.
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
  - more on this later.