Log-Based Recovery

- A log is a sequence of log records. The records keep information about update activities on the database.
  - The log is kept on stable storage
- When transaction $T_i$ starts, it registers itself by writing a $<T_i \text{ start}>$ 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 its last statement, the log record $<T_i \text{ commit}>$ is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification.
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.
  - (Will see later that how to postpone log record output to some extent.)
- Output of updated blocks to disk 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.

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 Example

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

- Note: \(B_X\) denotes block containing \(X\).

Recovery

- Previous example – what happens if failure in between (or during) some of the writes
  - Commit hasn’t been written, so transaction hasn’t happened
- *Undo* logging (Ariadne \(\sim 900\) BCE)
  - Restore state to before uncommitted transaction started
  - Log contains information on old state
  - “Follow the string” back to before you started
Write-Ahead Logging

• Logging must happen before write
  – Log captures state before write
  – Undo either restores written block to previous state, or contains same value if write hasn’t occurred

• When commit happens
  – All blocks modified by transaction
  – Commit \( T_i \) written to log

• When undoing
  – If commit \( T_i \) in log, then ignore log entries for \( T_i \)

*Undo logging (Immediate modification)*

\[ \begin{align*}
  T_1: & \quad \text{Read (A,} t)\; ; \ t \leftarrow t \times 2 \quad A = B \\
  & \quad \text{Write (A,} t)\; ; \\
  & \quad \text{Read (B,} t)\; ; \ t \leftarrow t \times 2 \\
  & \quad \text{Write (B,} t)\; ; \\
  & \quad \text{Output (A)}; \\
  & \quad \text{Output (B)};
\end{align*} \]
One “complication”

- Log is first written in memory
- Not written to disk on every action

memory

A: 8 16
B: 8 16
Log:
<T1,start>
<T1, A, 8>
<T1, B, 8>

BAD STATE
# 1

A: 8 16
B: 8

memory

A: 8 16
B: 8 16
Log:
<T1,start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>

BAD STATE
# 2

A: 8 16
B: 8

:.  
<T1, B, 8>
<T1, commit>
Undo logging rules

1. For every action generate undo log record (containing old value)
2. Before $x$ is modified on disk, log records pertaining to $x$ must be on disk (write ahead logging: WAL)
3. Before commit is flushed to log, all writes of transaction must be reflected on disk

Recovery rules: Undo logging

- For every $T_i$ with $<T_i, \text{start}>$ in log:
  - If $<T_i, \text{commit}>$ or $<T_i, \text{abort}>$ in log, do nothing
  - else For all $<T_i, X, v>$ in log:
    - write $(X, v)$
    - output $(X)$
    - Write $<T_i, \text{abort}>$ to log

***IS THIS CORRECT??***
Recovery rules: Undo logging

1. Let $S$ = set of transactions with $<T_i, \text{start}>$ in log, but no $<T_i, \text{commit}>$ (or $<T_i, \text{abort}>$) record in log

2. For each $<T_i, X, v>$ in log, in reverse order (latest $\rightarrow$ earliest) do:
   - if $T_i \in S$ then
     - write $(X, v)$
     - output $(X)$

3. For each $T_i \in S$ do
   - write $<T_i, \text{abort}>$ to log

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 $T_1$ updates A, then $T_2$ updates A and commits, and finally $T_1$ 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 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 $<T_i\text{abort}>$ 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

Recovering from Failure

- 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 either the record $<T_i\text{commit}>$ or $<T_i\text{abort}>$.
  - Transaction $T_i$ needs to be redone if the log
    - Contains the records $<T_i\text{start}>$
    - And contains the record $<T_i\text{commit}>$ or $<T_i\text{abort}>$
Recovering from Failure (Cont.)

- Suppose that transaction $T_i$ was undone earlier and the $\langle T_i, \text{abort} \rangle$ record was written to the log, and then a failure occurs,
- On recovery from failure transaction $T_i$ is redone
  - Such a redo redoes all the original actions of transaction $T_i$ 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.

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

Recovery actions in each case above are:
(a) undo ($T_0$): B is restored to 2000 and A to 1000, and log records $\langle T_0, B, 2000 \rangle$, $\langle T_0, A, 1000 \rangle$, $\langle T_0, \text{abort} \rangle$ 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 $\langle T_1, C, 700 \rangle$, $\langle T_1, \text{abort} \rangle$ 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
Redo logging (deferred modification)

T1: Read(A,t); t \times 2; write (A,t);  
Read(B,t); t \times 2; write (B,t);  
Output(A); Output(B)

Redo logging rules

1. For every action, generate redo log record (containing new value)
2. Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk
3. Flush log at commit
Recovery rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:
    
    \[
    \begin{align*}
    &\text{Write}(X, v) \\
    &\text{Output}(X)
    \end{align*}
    \]

**IS THIS CORRECT??**

---

Recovery rules: Redo logging

1. Let \( S = \) set of transactions with <Ti, commit> in log
2. For each <Ti, X, v> in log, in forward order (earliest \( \rightarrow \) latest) do:
   - if Ti \( \in S \) then
     \[
     \begin{align*}
     &\text{Write}(X, v) \\
     &\text{Output}(X) \quad \text{optional}
     \end{align*}
     \]
Recovery is very, very SLOW!

Redo log:

First Record (1 year ago) T1 wrote A,B
Last Record Committed a year ago

--> STILL, Need to redo after crash!!

Solution: Checkpoint (simple version)

• Periodically:
  1. Do not accept new transactions
  2. Wait until all transactions finish
  3. Flush all log records to disk (log)
  4. Flush all buffers to disk (DB) (do not discard buffers)
  5. Write “checkpoint” record on disk (log)
  6. Resume transaction processing
Example: what to do at recovery?

- Redo log (disk):

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Crash</td>
</tr>
<tr>
<td></td>
<td>&lt;T1,A,16&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;T1,commit&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checkpoint</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;T2,B,17&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;T2,commit&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T3,C,21&gt;</td>
</tr>
</tbody>
</table>

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 $T_i$.
  - Scan backwards from end of log to find the most recent $<\text{checkpoint } L>$ 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
  - Continue scanning backwards till a record $<T_i, \text{start}>$ is found for every transaction $T_i$ in $L$.
  - Parts of log prior to earliest $<T_i, \text{start}>$ 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

Remaining drawbacks:

- *Undo logging*: cannot bring backup DB copies up to date
- *Redo logging*: need to keep all modified blocks in memory until commit
Solution: undo/redo logging!

Update ⇒ <Ti, Xid, New X val, Old X val>
page X

Rules

- Page X can be flushed before or after Ti commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)
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

After undo phase completes, normal transaction processing can commence
Example of Recovery

Beginning of log

- $T_0$ start
- $T_0$, B, 2000, 2050
- $T_1$ start
- $<\text{checkpoint } \{T_0, T_1\}>$
- $T_1$, C, 700, 600
- $T_1$ commit
- $T_2$ start
- $T_2$, A, 500, 400
- $T_0$, B, 2000
- $T_0$ abort
- $T_2$ abort

End of log at crash!

Start log records found for all transactions in undo list

Redo Pass

Undo Pass

$T_0$ rollback (during normal operation) begins

$T_0$ rollback complete

$T_2$ is incomplete at crash

Undo list: $T_2$

$T_2$ rolled back in undo pass

Non-quiesce checkpoint

LOG

for undo

dirty buffer
pool pages
flushed

Start-ckpt
active TR: $T_1, T_2, ...$

end ckpt

...
Examples: What to do at recovery time?

no T1 commit

LOG

_undo T1 (undo a,b)

Example

LOG

_redo T1: (redo b,c)
Recovery process:

- **Backwards pass** (end of log ⇔ latest checkpoint start)
  - construct set $S$ of committed transactions
  - undo actions of transactions not in $S$
- **Undo pending transactions**
  - follow undo chains for transactions in (checkpoint active list) - $S$
- **Forward pass** (latest checkpoint start ⇔ end of log)
  - redo actions of $S$ transactions