ACID Properties of DB Transaction

- Atomicity
- Consistency
- Isolation
- Durability

Failure Recovery

- Ensure atomicity and durability despite system failures

```sql
start transaction;
select balance from accounts where id=1;
update accounts set balance=balance+100
where id=1;
update accounts set balance=balance-100
where id=2;
commit;
```

Failure Model

- System crash
  - CPU halts
  - Data in memory is lost
  - Data on disk is OK
- Everything else

Logging

- Log
  - A sequence of log records
  - Append only

What Do We Log

Transaction → Log

```sql
start transaction;
select balance from accounts where id=1;
update accounts set balance=balance+100
where id=1;
update accounts set balance=balance+100
where id=2;
commit;
```
Log Records in SimpleDB

Record Type | Transaction # |
---|---|
<START, 27> | |
<SETINT, 27, accounts.tbl, 0, 38, 1000, 900> | |
<SETINT, 27, accounts.tbl, 2, 64, 10, 110> | |
<COMMIT, 27> | |

File Name | Block # | Position | Old Value | New Value
---|---|---|---|---

General Notation for Log Records

- `<START, T>`
- `<UPATE, T, X, v_x, v'_x>`
- `<COMMIT, T>`
- `<ABORT, T>`

Recover from System Crash

- Remove changes made by uncommitted transactions – Undo
- Reapply changes made by committed transactions – Redo

Recover with Undo Only

- Assumption: all changes made by committed transactions have been saved to disk

Example: Create Undo Logging Records

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Transaction;</td>
<td>&lt;START, T&gt;</td>
</tr>
<tr>
<td>Write(X, v'_X)</td>
<td>&lt;UPDATE, T, X, v'_X&gt;</td>
</tr>
<tr>
<td>Write(Y, v'_Y)</td>
<td>&lt;UPDATE, T, Y, v'_Y&gt;</td>
</tr>
<tr>
<td>Commit;</td>
<td>&lt;COMMIT, T&gt;</td>
</tr>
</tbody>
</table>

About Logging

- Undo logging records do not need to store the new values
  - Why??
- The key of logging is to decide when to flush to disk
  - The changes made by the transaction
  - The log records
Example: Flushing for Undo Recovery

- Order the actions, including $\text{Flush}(X)$ and $\text{Flush}(\log)$, into a sequence that allows Undo Recovery

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Transaction;</td>
<td>$\langle \text{START}, T \rangle$</td>
</tr>
<tr>
<td>Write($X$, $v$)</td>
<td>$\langle \text{UPDATE}, T, X, v \rangle$</td>
</tr>
<tr>
<td>Write($Y$, $v$)</td>
<td>$\langle \text{UPDATE}, T, Y, v \rangle$</td>
</tr>
<tr>
<td>Commit;</td>
<td>$\langle \text{COMMIT}, T \rangle$</td>
</tr>
</tbody>
</table>

Order Flush($X$) and Flush($\langle \text{UPDATE}, X \rangle$) for Undo

- Consider an incomplete transaction
  - (a) Both $X$ and $\langle \text{UPDATE}, X \rangle$ are written to disk
  - (b) $X$ is written to disk but not $\langle \text{UPDATE}, X \rangle$
  - (c) $\langle \text{UPDATE}, X \rangle$ is written to disk but not $X$
  - (d) Neither is written to disk

Write-Ahead Logging

- A modified buffer can be written to disk only after all of its update log records have been written to disk

Implement Write-Ahead Logging

- Each log record has a unique id called log sequence number (LSN)
- Each buffer page keeps the LSN of the log record corresponding to the latest change
- Before a buffer page is flushed, notify the log manager to flush the log up to the buffer’s LSN

Order Flush($\langle \text{COMMIT}, T \rangle$) for Undo

- $\langle \text{COMMIT}, T \rangle$ cannot be written to disk before new value of $X$ is written to disk
- Commit statement cannot return before $\langle \text{COMMIT}, T \rangle$ is written to disk

Undo Logging

- Write $\langle \text{UPDATE}, T, X, v, x \rangle$ to disk before writing new value of $X$ to disk
- Write $\langle \text{COMMIT}, T \rangle$ after writing all new values to disk
- $\text{COMMIT}$ returns after writing $\langle \text{COMMIT}, T \rangle$ to disk
Undo Recovery

- Scan the log
  - *Forward or backward?*
- `<COMMIT,T>`: add T to a list of committed transactions
- `<UPDATE,T,X,v_x>`: if T is not in the lists of committed transactions, restore X’s value to v_x

About Undo Recovery

- No need to keep the new value
- Scan the log once for recovery
- COMMIT must wait until all changes are flushed
- Idempotent – recovery processes can be run multiple times with the same result

Example: Flushing for Redo Recovery

- Order the actions, including `Flush(X)` and `Flush(<log>)`, into a sequence that allows Redo Recovery

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Transaction;</td>
<td><code>&lt;START, T&gt;</code></td>
</tr>
<tr>
<td>Write(X, v_x)</td>
<td><code>&lt;UPDATE, T, X, v_x&gt;</code></td>
</tr>
<tr>
<td>Write(Y, v_y)</td>
<td><code>&lt;UPDATE, T, Y, v_y&gt;</code></td>
</tr>
<tr>
<td>Commit;</td>
<td><code>&lt;COMMIT, T&gt;</code></td>
</tr>
</tbody>
</table>

Undo Logging and Recovery Example

- Consider two transactions T_1 and T_2
  - T_1 updates X and Y
  - T_2 updates Z
- Show a possible sequence of undo logging
- Discuss possible crushes and recoveries

Recover with Redo Only

- Assumption: *none* of the changes made by *uncommitted* transactions have been saved to disk

Redo Logging

- Write `<UPDATE,T,X,v_y>` and `<COMMIT,T>` to disk *before* writing *any* new value of the transaction to disk
- COMMIT returns *after* writing `<COMMIT,T>` to disk
Redo Recovery
- Scan the log to create a list of committed transactions
- Scan the log again to replay the updates of the committed transactions
  - Forward or backward??

About Redo Recovery
- A transaction must keep all the blocks it needs pinned until the transaction completes – increases buffer contention

Combine Undo and Redo – Undo/Redo Logging
- Write $<\text{UPDATE}, T, x, v, v'>$ to disk before writing new value of $X$ to disk
- COMMIT returns after writing $<\text{COMMIT}, T>$ to disk

Undo/Redo Recovery
- Stage 1: undo recovery
- Stage 2: redo recovery

Advantages of Undo/Redo
- Vs. Undo??
- Vs. Redo??

Checkpoint
- Log can get very large
- A recovery algorithm can stop scanning the log if it knows
  - All the remaining records are for completed transactions
  - All the changes made by these transactions have been written to disk
Quiescent Checkpointing

- Stop accepting new transactions
- Wait for all existing transactions to finish
- Flush all dirty buffer pages
- Create a <CHECKPOINT> log record
- Flush the log
- Start accepting new transactions

Nonquiescent Checkpointing

- Stop accepting new transactions
- Let $T_1, \ldots, T_k$ be the currently running transactions
- Flush all modified buffers
- Write the record $<$NQCKPT, $T_1, \ldots, T_k>$ to the log
- Start accepting new transactions

Quiescent vs. Nonquiescent

<table>
<thead>
<tr>
<th>Quiescent</th>
<th>Nonquiescent</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;START, 0&gt;</td>
<td>&lt;START, 0&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&lt;START, 1&gt;</td>
<td>&lt;START, 1&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&lt;COMMIT, 0&gt;</td>
<td>&lt;NQCKPT, 0, 1&gt;</td>
</tr>
<tr>
<td>...</td>
<td>&lt;START, 2&gt;</td>
</tr>
<tr>
<td>&lt;COMMIT, 1&gt;</td>
<td>&lt;COMMIT, 0&gt;</td>
</tr>
<tr>
<td>...</td>
<td>&lt;COMMIT, 1&gt;</td>
</tr>
<tr>
<td>&lt;START, 2&gt;</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Example: Nonquiescent Checkpoint

Using Undo/Redo Recovery

- <START, 0>
- <WRITE, 0, A, v_0, v_1>
- <START, 1>
- <START, 2>
- <COMMIT, 1>
- <WRITE, 2, B, v_0, v_2>
- <NQCKPT, 0, 2>
- <WRITE, 0, C, v_0, v_3>
- <COMMIT, 0>
- <START, 3>
- <WRITE, 2, D, v_0, v_4>
- <WRITE, 3, E, v_0, v_5>

About Nonquiescent Checkpointing

- Do not need to wait for existing transactions to complete
- *But why do we need to stop accepting new transactions?*
- Recovery algorithm may stop at
  - <NQCKPT> if all $\{T_1, \ldots, T_k\}$ committed, or
  - <START> of the earliest uncommitted transaction in $\{T_1, \ldots, T_k\}$

Readings

- Textbook
  - Chapter 13.1-13.3
  - Chapter 14.1-14.3
- SimpleDB source code
  - simpledb.log
  - simpledb.tx
  - simpledb.txt.recovery