Failure Recovery

- Ensure atomicity and durability despite system failures

```
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;
```

Acid Properties of DB Transaction

- Atomicity
- Consistency
- Isolation
- Durability

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

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

Everything else
Log Records in SimpleDB

<table>
<thead>
<tr>
<th>Record Type</th>
<th>Transaction #</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;START, 27&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;SETINIT, 27, accounts.tbl, 0, 38, 1000&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;SETINIT, 27, accounts.tbl, 2, 64, 10&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;COMMIT, 27&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File Name</th>
<th>Block #</th>
<th>Position</th>
<th>Old Value</th>
</tr>
</thead>
</table>

General Notation for Log Records

- `<START, T>`
- `<UPDATE, T, X, v, v' >`
- `<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

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

Example: Create Undo Logging Records

Transaction

- Start Transaction;
- Write(X, v,)
- Write(Y, v,)
- Commit;

Log

- `<START, T>`
- `<UPDATE, T, X, v>`
- `<UPDATE, T, Y, v>`
- `<COMMIT, T>`

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>(&lt;\text{START}, T&gt;&gt;</td>
</tr>
<tr>
<td>(\text{Write}(X, v_x))</td>
<td>(&lt;\text{UPDATE}, T, X, v_x&gt;&gt;</td>
</tr>
<tr>
<td>(\text{Write}(Y, v_y))</td>
<td>(&lt;\text{UPDATE}, T, Y, v_y&gt;&gt;</td>
</tr>
<tr>
<td>Commit;</td>
<td>(&lt;\text{COMMIT}, T&gt;&gt;</td>
</tr>
</tbody>
</table>

About the Actions

- \(\text{Write}(X, v_x)\) Update \(X\) in memory (i.e. buffer)
- \(\text{Flush}(X)\) Flush the buffer page that contains \(X\).
- \(<\text{UPDATE}, T, X, v_x>\>\) Create a log record in memory – log records have their own buffer page.
- \(\text{Flush}(\log)\) Flush the log buffer page. Note that all log records in the log buffer will be flushed to disk.

Order \(\text{Flush}(X)\) and \(\text{Flush}(<\text{UPDATE}, X>)\) for Undo

- Consider an incomplete transaction
  - (a) Both \(X\) and \(<\text{UPDATE}, X>\) are written to disk
  - (b) \(X\) is written to disk but not \(<\text{UPDATE}, X>\)
  - (c) \(<\text{UPDATE}, X>\) 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 \(\text{Flush}(<\text{COMMIT}, T>)\) for Undo

- \(<\text{COMMIT}, T>\) cannot be written to disk before new value of \(X\) is written to disk
- Commit statement cannot return before \(<\text{COMMIT}, T>\) is written to disk
Undo Logging

- Write \(<\text{UPDATE}, T, X, v_x, >\) to disk \textit{before} writing new value of X to disk
- Write \(<\text{COMMIT}, T, >\) \textit{after} writing all new values to disk
- COMMIT returns \textit{after} writing \(<\text{COMMIT}, T, >\) to disk

Undo Recovery

- Scan the log
  - \textit{Forward or backward??}
- \(<\text{COMMIT}, T, >\) : add T to a list of committed transactions
- \(<\text{UPDATE}, T, X, v_x, >\) : if T is not in the lists of committed transactions, restore X's value to \(v_x\)

Undo Logging and Recovery Example

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

About Undo Recovery

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

Recover with Redo Only

- Prerequisite: \textit{none} of the changes made by uncommitted transactions have been saved to disk

Example: Flushing for Redo Recovery

- Order the actions, including \texttt{Flush(X)} and \texttt{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>(&lt;\text{START}, T, &gt;)</td>
</tr>
<tr>
<td>Write(X, v_x')</td>
<td>(&lt;\text{UPDATE}, T, X, v_x', &gt;)</td>
</tr>
<tr>
<td>Write(Y, v_y')</td>
<td>(&lt;\text{UPDATE}, T, Y, v_y', &gt;)</td>
</tr>
<tr>
<td>Commit;</td>
<td>(&lt;\text{COMMIT}, T, &gt;)</td>
</tr>
</tbody>
</table>
Redo Logging
- Write \(<\text{UPDATE}, T, X, v, v'\rangle\) and \(<\text{COMMIT}, T\rangle\) to disk before writing any new value of the transaction to disk
- COMMIT returns after writing \(<\text{COMMIT}, T\rangle\) 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
- COMMIT can return after all log records are flushed – transactions complete faster than using Undo-only
  - Why??
- 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 \(<\text{UPDATE}, T, X, v, v'\rangle\) to disk before writing new value of X to disk
- COMMIT returns after writing \(<\text{COMMIT}, T\rangle\) 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
- An *Undo/Redo* 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 $T_1, ..., T_k$ be the currently running transactions
- Flush all modified buffers
- Write the record `<NQCKPT, T_1, ..., 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><code>&lt;START, 0&gt;</code></td>
<td><code>&lt;START, 0&gt;</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>&lt;START, 1&gt;</code></td>
<td><code>&lt;START, 1&gt;</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>&lt;COMMIT, 0&gt;</code></td>
<td><code>&lt;NQCKPT, 0, 1&gt;</code></td>
</tr>
<tr>
<td>...</td>
<td><code>&lt;START, 2&gt;</code></td>
</tr>
<tr>
<td><code>&lt;CHPT&gt;</code></td>
<td><code>&lt;COMMIT, 0&gt;</code></td>
</tr>
<tr>
<td><code>&lt;START, 2&gt;</code></td>
<td><code>&lt;COMMIT, 1&gt;</code></td>
</tr>
</tbody>
</table>
| ... | ...

Example: Nonquiescent Checkpoint

- Using Undo/Redo Recovery

  ```
  <START, 0>
  <WRITE, 0, A, v_0, v_0>
  <START, 1>
  <WRITE, 2, 0, v_0, v_0>
  <COMMIT, 1>
  <WRITE, 2, 0, v_0, v_0>
  <NQCKPT, 0, 2>
  <WRITE, 0, C, v_0, v_0>
  <COMMIT, 0>
  <START, 3>
  <WRITE, 2, 0, v_0, v_0>
  <WRITE, 3, E, v_0, v_0>
  ```

About Nonquiescent Checkpointing

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

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