ACID Properties of DB Transaction

- Atomicity
- Consistency
- Isolation
- Durability

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;
```

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
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

<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;SETINT, 27, accounts.tbl, 0, 38, 1000, 900&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;SETINT, 27, accounts.tbl, 2, 64, 10, 110&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;COMMIT, 27&gt;</td>
<td></td>
</tr>
</tbody>
</table>

General Notation for Log Records

- <START, T>
- <UPATE, T, X, v, v’>
- <COMMIT, T>
- <ABORT, T>

Record Type | Transaction # | File Name | Block # | Position | Old Value | New Value
-------------|---------------|-----------|---------|----------|-----------|----------

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, )</td>
<td>&lt;UPDATE, T, X, v, &gt;</td>
</tr>
<tr>
<td>Write(Y, v, )</td>
<td>&lt;UPDATE, T, Y, v, &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

Order the actions, including \texttt{Flush(X)} and \texttt{Flush(<log>)}, into a sequence that allows Undo Recovery

\begin{tabular}{|c|c|}
\hline
Transaction & Log \\
\hline
Start Transaction; Write(X, v_x) Write(Y, v_y) Commit; & \texttt{<START, T>} \\
\texttt{<UPDATE, T, X, v_x>} & \texttt{<UPDATE, T, Y, v_y>} \\
\texttt{<COMMIT, T>} & \\
\hline
\end{tabular}

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

\begin{itemize}
\item Consider an incomplete transaction
\item (a) Both \texttt{X} and \texttt{<UPDATE,X> are written to disk}
\item (b) \texttt{X} is written to disk but not \texttt{<UPDATE,X>}
\item (c) \texttt{<UPDATE,X> is written to disk but not X}
\item (d) Neither is written to disk
\end{itemize}

Write-Ahead Logging

\begin{itemize}
\item A modified buffer can be written to disk only after all of its update log records have been written to disk
\end{itemize}

Implement Write-Ahead Logging

\begin{itemize}
\item Each log record has a unique id called \textit{log sequence number} (LSN)
\item Each buffer page keeps the LSN of the log record corresponding to the latest change
\item Before a buffer page is flushed, notify the log manager to flush the log up to the buffer’s LSN
\end{itemize}

Order \texttt{Flush(<COMMIT,T>) for Undo}

\begin{itemize}
\item \texttt{<COMMIT,T>} cannot be written to disk before new value of \texttt{X} is written to disk
\item Commit statement cannot return before \texttt{<COMMIT,T>} is written to disk
\end{itemize}

Undo Logging

\begin{itemize}
\item Write \texttt{<UPDATE,T,X,v_x>} to disk before writing new value of \texttt{X} to disk
\item Write \texttt{<COMMIT,T>} after writing all new values to disk
\item COMMIT returns after writing \texttt{<COMMIT,T>} to disk
\end{itemize}
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 $v_x$

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

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

Recover with Redo Only

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

Example: Flushing for Redo Recovery

Order the actions, including $\text{Flush}(X)$ and $\text{Flush}(<\text{log}>)$, 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><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>

Redo Logging

- Write `<UPDATE,T,X,v_x,>` 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 $T_1, ..., T_k$ 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>&lt;START, 0&gt;</td>
<td>&lt;START, 0&gt;</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;COMMIT, 0&gt;</td>
</tr>
<tr>
<td>&lt;COMMIT, 1&gt;</td>
<td>&lt;COMMIT, 1&gt;</td>
</tr>
<tr>
<td>&lt;CHECKPOINT&gt;</td>
<td>&lt;NQCKPT, 0, 1&gt;</td>
</tr>
<tr>
<td>&lt;START, 2&gt;</td>
<td>&lt;START, 2&gt;</td>
</tr>
<tr>
<td>...</td>
<td>&lt;COMMIT, 0&gt;</td>
</tr>
<tr>
<td>...</td>
<td>&lt;COMMIT, 1&gt;</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_0> |
  | <START, 1> | <WRITE, 2, B, v_2, v_2> |
  | <COMMIT, 1> | <NQCKPT, 0, 2> |
  | <WRITE, 0, C, v_0, v_0> |
  | <COMMIT, 0> | <WRITE, 2, D, v_0, v_0> |
  | <START, 3> | <WRITE, 3, E, v_0, v_0> |
  | <WRITE, 2, D, v_0, v_0> |
  | ... | ... |

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 does not need to look beyond the start record of the earliest uncommitted transaction in $\{T_1, ..., T_k\}$

Failure Recovery in SimpleDB

- Log Manager
  - simpledb.log
- Recovery Manager
  - simpledb.tx.recovery
SimpleDB Log Manager

- Default log file: `simpledb.log`
- Grows the log one block at a time
- The last block is kept in memory (i.e. only needs one page)

Append()

- Records are treated as arrays of objects (String or int)
- A new block is created if the current block does not have enough room to hold the new record
- The LSN of a log record is the block number

Locate Records in a Block

Two records: `<1, "Hi">, <2, 32>`

<table>
<thead>
<tr>
<th></th>
<th>24</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

LogIterator

- LogIterator iterates through a log backwards
- Again, only keeps one block in memory
- BasicLogRecord is simply a page and the starting position of a record in the page – it's up to the Recovery Manager to decide how to read the record

SimpleDB Recovery Manager

- Each transaction has its own recovery manager

<table>
<thead>
<tr>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency Manager</td>
</tr>
<tr>
<td>Recovery Manager</td>
</tr>
</tbody>
</table>

LogRecord Interface

- Record types
  - Checkpoint (quiescent)
  - Start
  - Commit
  - Rollback
  - SetInt
  - SetString

- Record operations
  - Write to log
  - Get record type
  - Get transaction #
  - Undo
  - [Redo]
Log Record Format

- Array of Integer and String
  - Record type
  - Additional information (optional)
- See the `writeToLog()` method in each log record class

LogRecordIterator

- Built on top of LogIterator
- Convert each BasicLogRecord to an a LogRecord object

Example: LogViewer

- Display the log
  - Up to the last <CHECKPOINT>

Recovery Manager

- Each transaction operation (e.g. start, commit, setint, setstring, rollback) creates a log record
- Rollback: undo the changes made by this transaction
- Recovery: perform recovery for the whole database

Undo Recovery in SimpleDB

- Recovery is done inside a transaction
- Iterate through the log backward
  - EOF or <Checkpoint>: stop
  - <Commit> or <Abort>: add transaction number to a list of finished transactions
  - Other: if the transaction # is not in the list of finished transactions, call undo()
- Save the changes (i.e. flush buffers)
- Write a <Checkpoint> log record

Example: TestLogWriter

- Write some records in the log for testing purpose
Readings

- **Textbook**
  - Chapter 13.1-13.3
  - Chapter 14.1-14.3

- **SimpleDB source code**
  - simpledb.log
  - simpledb.tx
  - simpledb.txt.recovery