Transaction Processing: Recovery

CPS 216
Advanced Database Systems

Announcements

- Homework #4 due in 7 days (April 23)
- Recitation session this Friday (April 18)
  - Homework #4 Q&A
- Project demo period starting in 12 days (April 28)
  - Sign-up sheet will be available next Monday
- Final exam in 17 days (May 1, 2-5pm)
  - More details will be available next Monday

Review

- ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability
  - Concurrency control
  - Recovery

Execution model

- Before it can be operated upon, disk-resident data must first be brought into memory
  - input($X$): copy the disk block containing object $X$ to memory
  - $v = \text{read}(X)$: read the value of $X$ into a local variable $v$
    - Execute input($X$) first if necessary
  - write($X$, $v$): write value $v$ to $X$ in memory
    - Execute input($X$) first if necessary
  - output($X$): write the memory block containing $X$ to disk

Failures

- System crashes in the middle of a transaction $T$; partial effects of $T$ were written to disk
  - How do we undo $T$ (atomicity)?
- System crashes right after a transaction $T$ commits; not all effects of $T$ were written to disk
  - How do we complete $T$ (durability)?
- Media fails; data on disk corrupted
  - How do we reconstruct the database (durability)?

Naïve approach

- Force: When a transaction commits, all writes of this transaction must be reflected on disk
  - Without force, if system crashes right after $T$ commits, effects of $T$ will be lost
  - Problem: Lots of random writes hurt performance
- No steal: Writes of a transaction can only be flushed to disk at commit time
  - With steal, if system crashes before $T$ commits but after some writes of $T$ have been flushed to disk, there is no way to undo these writes
  - Problem: Holding on to all dirty blocks requires lots of memory
Logging

- Log
  - Sequence of log records, recording all changes made to the database
  - Written to stable storage (e.g., disk) during normal operation
  - Used in recovery
- One change turns into two—bad for performance?
  - But writes are sequential (append to the end of log)
  - Can use dedicated disk(s) to improve performance

Undo logging

Basic idea

- Every time something is modified on disk, record its old value in the log
- If system crashes, undo the writes of partially executed transactions by restoring the old values

No steal?

- Not needed because we can undo; can flush dirty blocks before commit

Force?

- Still needed; otherwise effects of committed transactions can be lost after crash

Undo logging example

\( T_1 \) (balance transfer of $100 from \( A \) to \( B \))

\[
\begin{align*}
a &= \text{read}(A); a = a - 100; \\
\text{write}(A, a); \\
b &= \text{read}(B); b = b + 100; \\
\text{write}(B, b); \\
\text{output}(A); \\
\text{output}(B);
\end{align*}
\]

System crash

WAL

- Recap of the situation to be avoided
  - \( T_1 \) has not completed yet
  - \( A \) is modified on disk already
  - But there is no log record for \( A \)
  - Cannot undo the modification of \( A \)!

- Solution: WAL (Write-Ahead Logging)
  - Before any database object \( X \) is modified on disk, the log record pertaining to \( X \) must be flushed

Undo logging: normal operations

- For every write, generate an undo log record containing the old value being overwritten
  \( \langle T_p, X, \text{old}_value_{of}_X \rangle \)
  - Typically (assuming physical logging)
    - \( T_p \): transaction id
    - \( X \): physical address of \( X \) (block id, offset)
    - \text{old}_value_{of}_X: bits
  - Also log \( \langle T_p, \text{start} \rangle, \langle T_p, \text{commit} \rangle, \langle T_p, \text{abort} \rangle \)
  - WAL

- Force: before the commit log record of \( T_p \) is flushed, all writes of \( T_p \) must be reflected on disk
Undo logging: recovery
- Identify $U$, the set of active transactions at time of crash
  - Log contains $(T, \text{start})$, but neither $(T, \text{commit})$ nor $(T, \text{abort})$
- Process log backward (why?)
  - For each $(T, X, \text{old_value})$ where $T$ is in $U$, issue `write(X, old_value)`, `output(X)` (why flush?)
  - For each $T$ in $U$, append $(T, \text{abort})$ to the end of the log (why?)

Redo logging
Basic idea
- Every time something is modified on disk, record its new value in the log
- If system crashes, redo the writes of committed transactions and ignore those that did not commit

Force?
- Not needed because we can redo; commit does not trigger writing of dirty blocks
- Still needed; otherwise there is no way to roll back changes made by incomplete transactions

Redo logging example
$T_1$ (balance transfer of $100 from A to B)$

Redo logging: recovery
- Identify $C$, the set of all committed transactions (those with commit log records in log)
- Process log forward
  - For each $(T, X, \text{new_value})$ where $T$ is in $C$, issue `write(X, new_value)` (`output(X)` is optional; why?)
  - For each incomplete transaction $T$ (with neither commit nor abort log record), append $(T, \text{abort})$ to the end of the log

Additional issues with undo logging
- Failure during recovery?
  - No problem, run recovery procedure again
  - Undo is idempotent!
- Can you truncate log?
  - Yes, after a successful recovery
  - Or, truncate any prefix that contain no log records for active transactions

Redo logging: normal operations
- For every write, generate a redo log record containing the new value being written $(T, X, \text{new_value of } X)$
- Also log $(T, \text{start}), (T, \text{commit}), (T, \text{abort})$
- WAL
- No steal
  - Before the commit log record of $T$ is flushed, no writes of $T$ can be flushed to disk
  - Requires keeping all dirty blocks in memory before commit

Redo logging: recovery
- Identify $C$, the set of all committed transactions (those with commit log records in log)
- Process log forward
  - For each $(T, X, \text{new_value})$ where $T$ is in $C$, issue `write(X, new_value)` (`output(X)` is optional; why?)
  - For each incomplete transaction $T$ (with neither commit nor abort log record), append $(T, \text{abort})$ to the end of the log
Additional issues with redo logging

- Failure during recovery?
  - No problem—redo is idempotent!
- Extremely slow recovery process!
  - I transferred the balance last year…
- Can you truncate log?
  - No, unless …

Checkpointing

- Naïve approach
  - Stop accepting new transactions (lame!)
  - Finish all active transactions
  - Take a database dump
  - Now safe to truncate the redo log
- Alternative: fuzzy checkpointing (more later)

Summary of redo and undo logging

- Undo logging—immediate write
  - Force
    - Excessive disk I/Os
    - Imagine many small transactions updating the same block
- Redo logging—deferred write
  - No steal
    - High memory requirement
    - Imagine a big transaction updating many blocks

Logging taxonomy

<table>
<thead>
<tr>
<th></th>
<th>No steal</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force</td>
<td>No logging</td>
<td>Undo logging</td>
</tr>
<tr>
<td>No force</td>
<td>Redo logging</td>
<td>Undo/re redo logging</td>
</tr>
</tbody>
</table>

Undo/re redo logging: normal operations

- Log both old and new values
  \[ T_i, X, \text{old value of } X, \text{new value of } X \]
- Also log \( T_i, \text{start} \), \( T_i, \text{commit} \), \( T_i, \text{abort} \)
- WAL
- Steal: If chosen for replacement, modified memory blocks can be flushed to disk anytime
- No force: When a transaction commits, modified memory blocks are not forced to disk
  - Buffer manager has complete freedom!

Undo/re redo logging example

\[ T_1 \text{ (balance transfer of $100 from } A \text{ to } B) \]
\[ a = \text{read}(A); a = a - 100; \]
\[ \text{write}(A, a); \]
\[ b = \text{read}(B); b = b + 100; \]
\[ \text{write}(B, b); \]
\[ \text{commit}; \]

Disk

Memory

<table>
<thead>
<tr>
<th>A</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>500</td>
</tr>
</tbody>
</table>

Log

<table>
<thead>
<tr>
<th>Disk</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>800</td>
</tr>
<tr>
<td>B</td>
<td>400</td>
</tr>
</tbody>
</table>

Steal: can flush before commit

No force: can flush after commit

No restriction on when memory blocks can/should be flushed
Fuzzy checkpointing

- Determine $S$, the set of currently active transactions, and log \( \text{begin-checkpoint } S \)
- Write to disk all memory blocks currently dirty
  - Regardless whether they are written by committed or uncommitted transactions (but do follow WAL)
  - Blocks that become dirty after begin-checkpoint do not need to be flushed
- Log \( \text{end-checkpoint } \)
- Between begin and end, continue processing old and new transactions

Undo/redo logging: recovery

- Scan log from the last \( \text{start-checkpoint } S \) with a matching \( \text{end-checkpoint } \) to find
  - $C$, the set of committed transactions since last checkpoint
  - $A$, the set of active transactions at the time of crash
  - $C$ and $A$ may contain transactions that started after checkpoint
- Scan forward from that start-checkpoint to end of the log, and redo transactions in $C$
  - No need to look before start-checkpoint for redo (why?)
- Scan the log backward to undo transactions in $A$
  - May scan past start-checkpoint (why?)
  - Optimization: each log record stores a pointer to the previous log record for the same transaction; follow the pointer during undo
- Append \( T, \text{abort} \) to the log for each $T$ in $A$

Physical vs. logical logging

- Physical logging (what we have assumed so far)
  - Log before and after images of data
- Logical logging
  - Log operations (e.g., insert a row into a table)
  - Smaller log records
    - An insertion could cause rearrangement of things on disk
    - Or trigger hundreds of other events
  - Sometimes necessary
    - Assume row-level rather than page/block-level locking
    - Data might have moved to another block at time of undo!
  - Much harder to make redo/undo idempotent
    - See solution offered by ARIES

ARIES


- Same basic ideas: steal, no force, WAL
- Three phases: analysis, redo, undo
  - Repeats history (redo even incomplete transactions)
  - Better than our simple algorithm
    - CLR (Compensation Log Record) for transaction aborts
    - Redo/undo on an object is only performed when necessary → idempotency requirement lifted → logical logging supported
    - Each disk block records the LSN (log sequence number) of the last change
  - Can take advantage of a partial checkpoint
    - Recovery can start from any start-checkpoint, not necessarily one that corresponds to an end-checkpoint