Transaction Processing:
Recovery

CPS 216
Advanced Database Systems

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)?

Review
• ACID
  – Atomicity
  – Consistency
  – Isolation
  – Durability

Concurrency control
Recovery

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
• Hey, one change turns into two!
  – Isn’t it bad for performance?
  – But writes are sequential (append to the end of log)
  – Can use dedicated disk(s) to improve performance

Execution model

• input($X$): copy the disk block containing object $X$ to memory
• read($X, v$): 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

Undo logging
• Basic idea
  – Every time you modify something on disk, record its old value in the log
  – If system crashes, undo the writes of partially executed transactions by restoring the old values
Undo logging example

$T_1$ (balance transfer of $100 from A to B)$
read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);
output(A);
output(B);

Memory
A = 800
B = 400

Disk
A = 800 700
B = 400 500

Log
<T_1, start>
<T_1, A, 800>
<T_1, B, 400>
<T_1, commit>

Another technicality

$T_1$ (balance transfer of $100 from A to B)$
read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);
output(A);
output(B);

Memory
A = 800
B = 400

Disk
A = 800 700
B = 400 500

Log
<T_1, start>
<T_1, A, 800>
<T_1, B, 400>
<T_1, commit>

One technicality

$T_1$ (balance transfer of $100 from A to B)$
read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);
output(A);
output(B);

Memory
A = 800
B = 400

Disk
A = 800 700
B = 400 500

Log
<T_1, start>
<T_1, A, 800>
<T_1, B, 400>

System crash

Haven’t been flushed yet

Force

• Recap of the situation to be avoided
  – $T_1$ has committed (the log says so)
  – Not all effects of $T_1$ have been flushed disk
  – Because there is no redo information in the log, we cannot redo the rest of $T_1$
  – So perhaps we should try redo logging?
• Solution: force
  – Before the commit record of a transaction is flushed to log, all writes of this transaction must be reflected on disk

WAL

• Recap of the situation to be avoided
  – $T_i$ 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 rules

• For every write, generate undo log record containing the old value being overwritten
  <T_i, X, old_value_of_X>
  – Typically (assuming physical logging)
    • T_i: transaction id
    • X: physical address of X (block id, offset)
    • old_value_of_X: bits
• WAL
• Force
Recovery with an undo log

- 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($X, \text{old_value}$) output($X$)
  - For each $T$ in $U$, 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 example

$T_1$ (balance transfer of $100$ from $A$ to $B$)
read($A, a$); $a = a - 100$;
write($A, a$);
read($B, b$); $b = b + 100$;
write($B, b$);
output($A$);
output($B$);

Redo logging

- Basic idea
  - Every time you modify something on disk, record its new value (which you are writing)
  - If system crashes, redo the writes of committed transactions and ignore those that did not commit

One technicality

$T_1$ (balance transfer of $100$ from $A$ to $B$)
read($A, a$); $a = a - 100$;
write($A, a$);
read($B, b$); $b = b + 100$;
write($B, b$);
output($A$);
output($B$);

No steal

- Recap of the situation to be avoided
  - $T_1$ has not completed yet
  - $A$ is modified on disk already
  - There is a log record for $A$ (i.e., WAL is followed)
  - Because there is no undo information in that log record, we cannot undo the modification of $A$!
  - Maybe undo/redo combined?
- Solution: no steal
  - Writes can be flushed only at commit time
  - Requires keeping all dirty blocks in memory—other transactions cannot steal any memory blocks
Redo logging rules

- For every write, generate redo log record containing the new value being written \(<T,, X, new\_value\_of\_X>\).  
- Do not modify any database objects on disk before you have flushed all log records for this transaction (including the commit record)  
  - That is, WAL and no steal.

Checkpointing

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

  ➢ Fuzzy checkpointing  
  - Example later

Recovery with a redo log

- Identify C, the set of all committed transactions (those with commit log record)  
- Process log forward Why?  
  - For each \(<T, X, new\_value>\) where \(T\) is in C, issue write\((X, new\_value)\) Why is output\((X)\) unnecessary here?  
- For each incomplete transaction \(T\) (with neither commit nor abort log record), append \(<T, abort>\) to the end of the log.

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

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…

Logging taxonomy

Assuming each transaction modifies just one block and locking is at the block level

<table>
<thead>
<tr>
<th></th>
<th>force</th>
<th>no steal</th>
<th>steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>no force</td>
<td>redo logging</td>
<td>undo logging</td>
<td>undo/redo logging</td>
</tr>
</tbody>
</table>

Next!
Undo/redo logging

- Log both old and new values
  \(<T, X, old\_value\_of\_X, new\_value\_of\_X>\)
- 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!

Recovery: analysis and redo phase

- Need to determine \(U\), the set of active transactions at time of crash
- Scan log backward to find the last end-checkpoint record and follow the pointer to find the corresponding \(<\text{start-checkpoint } S>\)
- Initially, let \(U = S\)
- Scan forward from that start-checkpoint to end of the log
  – For a log record \(<T, \text{start}>\), add \(T\) to \(U\)
  – For a log record \(<T, \text{commit } \mid \text{abort}>\), remove \(T\) from \(U\)
  – For a log record \(<T, X, old, new>, \text{issue write}(X, new)>\)
  – Repeats history!

Undo/redo logging example

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

\[
\begin{align*}
\text{read}(A, a); & \quad a = a - 100; \\
\text{write}(A, a); & \quad A = 800 \quad 700 \\
\text{read}(B, b); & \quad b = b + 100; \\
\text{write}(B, b); & \quad B = 400 \quad 500
\end{align*}
\]

Memory
\[
\begin{array}{c|c}
A & 800 \quad 700 \\
B & 400 \quad 500
\end{array}
\]

Log
\[
\begin{align*}
<T_1, \text{start}> & \quad <T_1, A, 800, 700> \\
<T_1, B, 400, 500> & \quad <T_1, \text{commit}>
\end{align*}
\]

Fuzzy checkpointing

- Determine \(S\), the set of currently active transactions, and log \(<\text{begin-checkpoint } S>\)
- Flush all modified memory blocks at your leisure
  – Regardless whether they are written by committed or uncommitted transactions (but do follow WAL)
- Log \(<\text{end-checkpoint begin-checkpoint\_location}>\)
- Between begin and end, continue processing old and new transactions

Recovery: undo phase

- Scan log backward
  – Undo the effects of transactions in \(U\)
  – That is, for each log record \(<T, X, old, new>\) where \(T\) is in \(U\), issue write\((X, old)\), and log this operation too (part of the repeating-history paradigm)
  – Log \(<T, \text{abort}>\) when all effects of \(T\) have been undone
- An optimization
  – Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo
- Is it possible that undo overwrites the effect of a committed transaction?
  – Not if strict 2PL!

Physical versus 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
Selective redo?

- Possible optimization for our recovery procedure:
  - Selectively redo only committed transactions
  - Lots of algorithms do it (some even undo before redo)
- What is the catch?
  - $T_1.op_1, T_2.op_1, T_1.op_2 \ (T_1.commit)$
  - Repeating history: $T_1.op_1, T_2.op_2, T_1.op_2, \undo(T_2.op_2)$
    - Exactly the same as normal transaction abort
  - Selective redo: $T_1.op_1, T_1.op_2, \undo(T_2.op_1)$
    - What if $T_2.op_1$ produced some side effects that $T_1.op_2$ relies on?
    - Not possible with page-level locking and physical logging
    - In general hard to guarantee

ARIES

- Same basic ideas: steal, no force, WAL
- Three phases: analysis, redo, undo
  - Repeats history
- CLR (Compensation Log Record) for transaction aborts
- More efficient than our simple algorithm
  - Redo/undo on an object is only performed when necessary
    - Each disk block records the last writer
  - Can take advantage of a partial checkpoint
    - Recovery can start from any start-checkpoint, not necessarily one that corresponds to an end-checkpoint