Transaction Processing
Introduction to Databases
CompSci 316 Fall 2016

Announcements (Thu., Dec. 1)

• Homework #4 due next Tuesday
  • Except the last Gradiance problem (due Thursday)
• Project demos—please sign up via Google Doc!
  • Early in-class demos on 1/8
• Final exam Thur. Dec. 15 7-10pm
  • Different room: LSRC B101
    • Open-book, open-notes
    • Comprehensive, but with strong emphasis on the second half of the course
  • Sample final to be posted soon

Announcements (Tue., Dec. 6)

• Homework #4 due today
  • Except the last Gradiance problem (due Thursday)
• Project demos to start this Friday
  • Final schedule to be emailed soon
  • Nobody signed up for early in-class demo!
• Final exam Thur. Dec. 15 7-10pm
  • Different room: LSRC B101
    • Open-book, open-notes
    • Comprehensive, but with strong emphasis on the second half of the course
  • Sample final posted on Sakai (solution to be posed soon)
Review

• ACID
  • Atomicity: TX’s are either completely done or not done at all
  • Consistency: TX’s should leave the database in a consistent state
  • Isolation: TX’s must behave as if they are executed in isolation
  • Durability: Effects of committed TX’s are resilient against failures
• SQL transactions
  -- Begins implicitly
  SELECT …;
  UPDATE …;
  ROLLBACK | COMMIT;

Concurrency control

• Goal: ensure the “I” (isolation) in ACID

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{read(A)} & \text{read(A)} \\
\text{write(A)} & \text{write(A)} \\
\text{read(B)} & \text{read(C)} \\
\text{write(B)} & \text{write(C)} \\
\text{commit} & \text{commit} \\
\end{array}
\]

Good versus bad schedules

\[
\begin{array}{c|c|c|c|c|c|c}
T_1 & T_2 & T_1 & T_2 & T_1 & T_2 \\
\hline
r(A) & w(A) & r(A) & w(A) & r(A) & w(A) \\
r(B) & w(B) & r(A) & w(A) & r(B) & w(A) \\
\end{array}
\]

Good!
Serial schedule

- Execute transactions in order, with no interleaving of operations
  - $T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C)$
  - $T_3.r(A), T_3.w(A), T_3.r(B), T_3.w(B)$
- Isolation achieved by definition!

- Problem: no concurrency at all
- Question: how to reorder operations to allow more concurrency

Conflicting operations

- Two operations on the same data item conflict if at least one of the operations is a write
  - $r(X)$ and $w(X)$ conflict
  - $w(X)$ and $r(X)$ conflict
  - $r(X)$ and $r(X)$ do not conflict
  - $r/w(X)$ and $r/w(Y)$ do not conflict

- Order of conflicting operations matters
  - E.g., if $T_1.r(A)$ precedes $T_2.w(A)$, then conceptually, $T_1$ should precede $T_2$

Precedence graph

- A node for each transaction
- A directed edge from $T_i$ to $T_j$ if an operation of $T_i$ precedes and conflicts with an operation of $T_j$ in the schedule

Good: no cycle

Bad: cycle
Conflict-serializable schedule

- A schedule is conflict-serializable iff its precedence graph has no cycles.
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”):
  - In that serial schedule, transactions are executed in the topological order of the precedence graph.
  - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions.

Locking

- Rules:
  - If a transaction wants to read an object, it must first request a shared lock (S mode) on that object.
  - If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object.
  - Allow one exclusive lock, or multiple shared locks.

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Mode of the lock requested</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Compatibility matrix

Basic locking is not enough

- Possible schedule under locking:
  - Add 1 to both A and B (preserve \( A = B \)).
  - Multiply both A and B by 2 (preserves \( A = B \)).
- But still not conflict-serializable:
  - A ≠ B!
Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks

\[
\begin{array}{c|c|c|c}
\tau_1 & \tau_2 & \tau_3 & \tau_4 \\
\hline
\text{lock-}X(A) & \text{lock-}X(A) & \text{lock-}X(A) & \text{lock-}X(A) \\
\text{r(A)} & \text{w(A)} & \text{w(A)} & \text{r(A)} \\
\text{lock-}X(B) & \text{lock-}X(B) & \text{lock-}X(B) & \text{lock-}X(B) \\
\text{unlock(A)} & \text{unlock(A)} & \text{unlock(A)} & \text{unlock(A)} \\
\end{array}
\]

2PL guarantees a conflict-serializable schedule

\[
\begin{array}{c|c|c|c|c}
\tau_1 & \tau_2 & \tau_3 & \tau_4 \\
\hline
\text{lock-}X(A) & \text{lock-}X(A) & \text{lock-}X(A) & \text{lock-}X(A) \\
\text{r(A)} & \text{w(A)} & \text{w(A)} & \text{r(A)} \\
\text{lock-}X(B) & \text{lock-}X(B) & \text{lock-}X(B) & \text{lock-}X(B) \\
\text{unlock(A)} & \text{unlock(A)} & \text{unlock(A)} & \text{unlock(A)} \\
\hline
\text{lock-}X(A) & \text{lock-}X(A) & \text{lock-}X(A) & \text{lock-}X(A) \\
\text{r(B)} & \text{w(B)} & \text{w(B)} & \text{r(B)} \\
\text{lock-}X(B) & \text{lock-}X(B) & \text{lock-}X(B) & \text{lock-}X(B) \\
\text{unlock(B)} & \text{unlock(B)} & \text{unlock(B)} & \text{unlock(B)} \\
\end{array}
\]

Cannot obtain the lock on B until \( \tau_1 \) unlocks

Remaining problems of 2PL

- \( \tau_2 \) has read uncommitted data written by \( \tau_1 \)
- If \( \tau_1 \) aborts, then \( \tau_2 \) must abort as well
  - Cascading aborts possible if other transactions have read data written by \( \tau_2 \)
- Even worse, what if \( \tau_2 \) commits before \( \tau_1 \)?
  - Schedule is not recoverable if the system crashes right after \( \tau_2 \) commits

Strict 2PL

- Only release locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts
- Used in many commercial DBMS
  - Oracle is a notable exception
Recovery

• Goal: ensure “A” (atomicity) and “D” (durability)

Execution model

To read/write X
• The disk block containing X must be first brought into memory
• X is read/written in memory
• The memory block containing X, if modified, must be written back (flushed) to disk eventually

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)?
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:

- **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:

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—bad for performance?
  - But writes are sequential (append to the end of log)
  - Can use dedicated disk(s) to improve performance

Undo/redo logging rules

- When a transaction T starts, log (T\_start)
- Record values before and after each modification:
  - (T\_X, old_value\_of\_X, new_value\_of\_X)
  - T\_is transaction id and X identifies the data item
- A transaction T is committed when its commit log record (T\_commit) is written to disk
- **Write-ahead logging (WAL)**: Before X is modified on disk, the log record pertaining to X must be flushed.
  - Without WAL, system might crash after X is modified on disk but before its log record is written to disk—no way to undo
- **No force**: A transaction can commit even if its modified memory blocks have not been written to disk (since redo information is logged)
- **Steal**: Modified memory blocks can be flushed to disk anytime (since undo information is logged)
Undo/redo logging example

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

```plaintext
read(A, a);
write(A, a);
read(B, b);
write(B, b);
commit;
```

Memory buffer:

- $A = 800$
- $B = 400$

Disk:

- $A = 800$
- $B = 400$

Log:

- $(T_1, \text{start})$
- $(T_1, A, 800, 700)$
- $(T_1, B, 400, 500)$
- $(T_1, \text{commit})$

Steal: can flush before commit
No force: can flush after commit

No restriction (except WAL) on when memory blocks can/should be flushed

Checkpointering

- Where does recovery start?

  Naive approach:
  - To checkpoint:
    - Stop accepting new transactions (lame!)
    - Finish all active transactions
    - Take a database dump
  - To recover:
    - Start from last checkpoint

  ![Checkpointing Image](http://www.saintlouischeckpoints.com/wp-content/uploads/2013/08/dui20checkpoint200220172011.jpg)

Fuzzy checkpointing

- Determine $S$, the set of (ids of) currently active transactions, and log $(\text{begin-checkpoint } S)$
- Flush all blocks (dirty at the time of the checkpoint) at your leisure
- Log $(\text{end-checkpoint begin-checkpoint_location})$
- Between begin and end, continue processing old and new transactions
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 (start-checkpoint $S$)
• Initially, let $U$ be $S$
• 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 $(T, \text{commit | abort})$, remove $T$ from $U$
  • For a log record $(T, X, \text{old}, \text{new})$, issue write($X$, new)
    • Basically repeats history!

Recovery: undo phase

• Scan log backward
• Undo the effects of transactions in $U$
  • That is, for each log record $(T, X, \text{old}, \text{new})$ 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

Summary

• Concurrency control
  • Serial schedule: no interleaving
  • Conflict-serializable schedule: no cycles in the precedence graph; equivalent to a serial schedule
  • 2PL: guarantees a conflict-serializable schedule
  • Strict 2PL: also guarantees recoverability
• Recovery: undo/redo logging with fuzzy checkpointing
  • Normal operation: write-ahead logging, no force, steal
  • Recovery: first redo (forward), and then undo (backward)