Transaction Processing

Introduction to Databases

CompSci 316 Spring 2017
Announcements (Mon., Apr 3)

• Milestone2 feedback on piazza threads
Review from Lecture 12

• 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;
Review: SQL isolation levels

• Strongest isolation level: **SERIALIZABLE**
  • Mimics “complete isolation”
  • i.e. as if the transactions are executed one by one (serial schedule)
  • the executed schedule is equivalent to such a schedule (therefore is “serializable”)

• Weaker isolation levels:
  • **REPEATABLE READ**
  • **READ COMMITTED**
  • **READ UNCOMMITTED**

• Increase performance by eliminating overhead and allowing higher degrees of concurrency

• Trade-off: sometimes you get the “wrong” answer
Review: READ UNCOMMITTED

• Can read “dirty” data
  • A data item is dirty if it is written by an uncommitted transaction

• Problem: What if the transaction that wrote the dirty data eventually aborts?

• Example: wrong average
  • -- T1:  -- T2:
    UPDATE User
    SET pop = 0.99
    WHERE uid = 142;
    SELECT AVG(pop)
    FROM User;
    ROLLBACK;
    COMMIT;
Review: READ COMMITTED

- No dirty reads, but **non-repeatable reads** possible
  - Reading the same data item twice can produce different results

- Example: different averages
  - **T1:**
    ```sql
    UPDATE User
    SET pop = 0.99
    WHERE uid = 142;
    COMMIT;
    ```
  - **T2:**
    ```sql
    SELECT AVG(pop)
    FROM User;
    COMMIT;
    SELECT AVG(pop)
    FROM User;
    COMMIT;
    ```
Review: REPEATABLE READ

• Reads are repeatable, but may see **phantoms**
• Example: different average (still!)
  • -- T1:
  ```sql
  INSERT INTO User
  VALUES(789, 'Nelson', 10, 0.1);
  COMMIT;
  
  SELECT AVG(pop)
  FROM User;
  COMMIT;
  ```
  -- T2:
  ```sql
  SELECT AVG(pop)
  FROM User;
  COMMIT;
  ```
Next

Approaches to

• Concurrency Control (CC)
• Recovery
Concurrent control

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

\[ T_1: \]
read(A);
write(A);
read(B);
write(B);
commit;

\[ T_2: \]
read(A);
write(A);
read(C);
write(C);
commit;

Diagram:

\[ \begin{array}{ccc}
A & B & C \\
\end{array} \]
Good versus bad schedules

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td></td>
</tr>
<tr>
<td>w(A)</td>
<td></td>
</tr>
<tr>
<td>r(B)</td>
<td></td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
</tr>
<tr>
<td>r(C)</td>
<td></td>
</tr>
<tr>
<td>w(C)</td>
<td></td>
</tr>
</tbody>
</table>

**Good!**

**Bad!**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(C)</td>
<td>w(B)</td>
</tr>
<tr>
<td>w(C)</td>
<td></td>
</tr>
</tbody>
</table>

- **Read 400**
- **Write 400 – 100**

**Good! (But why?)**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td></td>
</tr>
<tr>
<td>r(B)</td>
<td></td>
</tr>
<tr>
<td>r(C)</td>
<td></td>
</tr>
<tr>
<td>w(C)</td>
<td></td>
</tr>
</tbody>
</table>

- **Read 400**
- **Write 400 – 50**
Serial schedule

• Execute transactions in order, with no interleaving of operations
  • $T_1.r(A)$, $T_1.w(A)$, $T_1.r(B)$, $T_1.w(B)$, $T_2.r(A)$, $T_2.w(A)$, $T_2.r(C)$, $T_2.w(C)$
  • $T_2.r(A)$, $T_2.w(A)$, $T_2.r(C)$, $T_2.w(C)$, $T_1.r(A)$, $T_1.w(A)$, $T_1.r(B)$, $T_1.w(B)$
  $\therefore$ 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
  • w(X) and w(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

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(C)</td>
</tr>
</tbody>
</table>

Good: no cycle

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(C)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(C)</td>
</tr>
</tbody>
</table>

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
what are the conflicts in this schedule?
## 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
  - no additional S lock needed for reading
- 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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td><strong>S</strong></td>
</tr>
<tr>
<td><strong>X</strong></td>
<td><strong>X</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Grant the lock?</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong> Yes</td>
</tr>
<tr>
<td><strong>X</strong> No</td>
</tr>
</tbody>
</table>
Basic locking is not enough

Add 1 to both A and B (preserve A=B)
Read 100
Write 100+1

Possible schedule under locking

But still not conflict-serializable!

Multiply both A and B by 2 (preserves A=B)

what are the conflicts?
try locking individual R/W actions

Possible schedule under locking

Basic locking is not enough

Add 1 to both A and B (preserve A=B)
Read 100
Write 100+1

Possible schedule under locking

But still not conflict-serializable!

Multiply both A and B by 2 (preserves A=B)

what are the conflicts?
try locking individual R/W actions

Add 1 to both A and B (preserve A=B)
Read 100
Write 100+1

Possible schedule under locking

But still not conflict-serializable!

Multiply both A and B by 2 (preserves A=B)

what are the conflicts?
try locking individual R/W actions
Two-phase locking (2PL)

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

2PL guarantees a conflict-serializable schedule

Cannot obtain the lock on B until $T_1$ unlocks
## Remaining problems of 2PL

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(B)</td>
</tr>
</tbody>
</table>

- **$T_2$** has read uncommitted data written by $T_1$
- If $T_1$ aborts, then $T_2$ must abort as well
- **Cascading aborts** possible if other transactions have read data written by $T_2$

- Even worse, what if $T_2$ commits before $T_1$?
  - Schedule is **not recoverable** if the system crashes right after $T_2$ 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

• Can create “deadlocks”
  • T1 is waiting for lock on B to be released by T2
  • T2 is waiting for lock on A to be released by T3
  • T3 is waiting for lock on C to be released by T1
  • to detect, can use “wait-for” graphs
  • to break, use timestamp as preference in a queue
Other approaches to CC

• Lock-based CC
  • SQLite, SQL Sever, DB2

• Multi-version CC (MVCC)
  • Create a “new version” for writing, read appropriate version
  • Postgres, Oracle

• Optimistic CC
  • validate before commit, if failed, roll back

• Time-stamp-based CC
  • Assign and update R/W timestamp of each object
  • See if “safe” to R/W
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: 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
FORCE/NO FORCE and STEAL/NO STEAL

- **Force** every write to disk?
  - Poor response time
  - But provides durability

- **Steal** buffer-pool frames from uncommitted transactions?
  - If not, poor throughput
  - If so, how can we ensure atomicity?
More on Steal and Force

• STEAL  (why enforcing Atomicity is hard)
  • To steal frame F: Current page in F (say P) is written to disk; some transaction holds lock on P
    • What if the transaction with the lock on P aborts?
    • Must remember the OLD value of P at steal time (to support UNDOing the write to page P)

• NO FORCE  (why enforcing Durability is hard)
  • What if system crashes before a modified page is written to disk?
    • Write as little as possible, in a convenient place, at commit time (i.e. remember the NEW value of P), to support REDOing modifications.
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
  - Any drawback?

- 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 $T_i$ starts, log $\langle T_i, \text{start} \rangle$

• Record values before and after each modification: $\langle T_i, X, \text{old\_value\_of\_X}, \text{new\_value\_of\_X} \rangle$
  • $T_i$ is transaction id and $X$ identifies the data item

• A transaction $T_i$ is committed when its commit log record $\langle T_i, \text{commit} \rangle$ 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 be 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 $B$)

read($A, a$); $a = a - 100$;
write($A, a$);
read($B, b$); $b = b + 100$;
write($B, b$);
commit;

No force: can flush after commit

Steal: can flush before commit

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

- Where does recovery start?
  
  Naïve approach:
  
  - To checkpoint:
    - Stop accepting new transactions (lame!)
    - Finish all active transactions
    - Take a database dump
  
  - To recover:
    - Start from last checkpoint
Fuzzy checkpointing

• Determine $S$, the set of (ids of) currently active transactions, and log $\langle \text{begin-checkpoint } S \rangle$

• Flush all blocks (dirty at the time of the checkpoint) at your leisure

• Log $\langle \text{end-checkpoint } \text{begin-checkpoint}_\text{location} \rangle$

• 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 < $T$, start $>$, add $T$ to $U$
  - For a log record < $T$, commit | abort $>$, remove $T$ from $U$
  - For a log record < $T$, $X$, old, new $>$, issue write($X$, new)

$\leftarrow$ Basically repeats history!
Recovery: undo phase

• Scan log **backward**
  • Undo the effects of transactions in \( U \)
  • That is, for each log record \( \langle T, X, \text{old}, \text{new} \rangle \) where \( T \) is in \( U \), issue \( \text{write}(X, \text{old}) \), and log this operation too (part of the “repeating-history” paradigm)
  • Log \( \langle T, \text{abort} \rangle \) 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

• This is the basic idea of “ARIES” protocol for UNDO/REDO log
  • Only UNDO (STEAL) or only REDO (NO FORCE) is possible too (see book)
Deadlock Detection Example

Example:

T1:  S(A), R(A), S(B)
T2:  X(B), W(B)  X(C)
T3:  S(C), R(C)  X(A)
T4: 

S(A): requesting shared lock on A
X(A): requesting exclusive lock on A
Wait-for graph
No violation of 2PL!
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)