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: 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:
    - UPDATE User SET pop = 0.99 WHERE uid = 142;
    - COMMIT;
    - SELECT AVG(pop) FROM User;
  - `~ T2:
    - SELECT AVG(pop) FROM User;

Review: REPEATABLE READ

- Reads are repeatable, but may see phantoms

- **Example:** different average (still)
  - `~ T1:
    - INSERT INTO User VALUES(789, 'Nelson', 10, 0.1);
    - COMMIT;
  - `~ T2:
    - SELECT AVG(pop) FROM User;

Next

Approaches to

• Concurrency Control (CC)
• Recovery

Concurrency control

• Goal: ensure the "I" (isolation) in ACID

Good versus bad schedules

<table>
<thead>
<tr>
<th>Good!</th>
<th>Bad!</th>
<th>Good! (But why?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>T₂</td>
<td>T₁</td>
</tr>
<tr>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td></td>
<td>r(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>w(A)</td>
<td>r(C)</td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Serial schedule

• Execute transactions in order, with no interleaving of operations

- T₁.r(A), T₁.w(A), T₁.r(B), T₁.w(B), T₂.r(A), T₂.w(A), T₂.r(C), T₂.w(C)
- T₂.r(A), T₂.w(A), T₂.r(C), T₂.w(C), T₁.r(A), T₁.w(A), T₁.r(B), T₁.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
  • 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 T₁.r(A) precedes T₂.w(A), then conceptually, T₁ should precede T₂

Precedence graph

• A node for each transaction
• A directed edge from Tᵢ to Tⱼ if an operation of Tᵢ precedes and conflicts with an operation of Tⱼ in the schedule

- T₁: read(A); write(A); read(B); write(B); write(C); commit;
- T₂: read(A); read(C); commit;
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.
  - No additional S lock needed for reading.
  - Allow one exclusive lock, or multiple shared locks.

Mode of lock(s) currently held by other transactions

- Compatibility matrix

Two-phase locking (2PL)

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

Conflicts in this schedule:

Example

- Add 1 to both A and B (preserves A=B).
- Multiply both A and B by 2 (preserves A=B).
- What are the conflicts?
- Try locking individual R/W actions.
- But still not conflict-serializable!
- Mode of the lock requested

Basic locking is not enough

- Locking is not enough to ensure serializability.
- Possible schedule under locking.
- But still not conflict-serializable!
- Grant the lock?
- Compatibility matrix

Remaining problems of 2PL

- The 2PL mode does not guarantee serializability.
- Multiple transactions may interfere with each other.
- Even worse, what if T2 commits before T1?
- Schedule is not recoverable if the system crashes right after T2 commits.

- T2 has read uncommitted data written by T1.
- If T1 aborts, then T2 must abort as well.
- Cascading aborts possible if other transactions have read data written by T2.

- Even worse, what if T2 commits before T1?
- Schedule is not recoverable if the system crashes right after T2 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

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
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 \( (T_i, \text{start}) \)
• Record values before and after each modification:
  \( (T_i, X, \text{old}_\text{value}_X, \text{new}_\text{value}_X) \)
  • \( T_i \) is transaction id and \( X \) identifies the data item
• A transaction \( T_i \) is committed when its commit log record
  \( (T_i, \text{commit}) \) 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 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)

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

\[
\begin{align*}
\text{Memory buffer} &; \\
A & = 800 \\
B & = 400
\end{align*}
\]

Disk

\[
\begin{align*}
\text{Disk} &; \\
A & = 700 \\
B & = 500
\end{align*}
\]

Log

\[
\begin{align*}
(T_1, \text{start}) &; \\
(T_1, A, 800, 700) &; \\
(T_1, B, 400, 500) &; \\
(T_1, \text{commit}) &;
\end{align*}
\]

Steal: can flush before commit
No force: can flush after 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 \( (\text{begin-checkpoint } S) \)
• Flush all blocks (dirty at the time of the checkpoint) at your leisure
• Log \( (\text{end-checkpoint } \text{begin-checkpoint}_\text{location}) \)
• 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 \( (\text{start-checkpoint } S) \)
• 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 \( (T, \text{commit} | \text{abort}) \), 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 \((T, X, \text{old, new})\) where \(T\) is in \(U\), issue write\((X, \text{old})\), 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
- 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)

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)