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

```plaintext
T_1: read(A);
write(A);
read(B);
write(B);
commit;

T_2: read(A);
write(A);
read(C);
write(C);
commit;
```
Good versus bad schedules

<table>
<thead>
<tr>
<th></th>
<th>T₂</th>
<th></th>
<th>T₂</th>
<th></th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
<td>r(B)</td>
<td>w(C)</td>
<td>w(C)</td>
<td>w(C)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
</tr>
</tbody>
</table>

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - T₁.r(A), T₁.w(A), T₁.r(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
  - r/w(X) and r/w(Y) do not
- Order of conflicting operations matters
  - 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 $T_i$ to $T_j$ if an operation of $T_i$ precedes and conflicts with an operation of $T_j$ in the schedule

\[
\begin{array}{cccc}
T_1 & T_2 & T_1 & T_2 \\
\text{r(A)} & \text{r(A)} & \text{r(A)} & \\
\text{w(A)} & \text{w(A)} & \text{w(A)} & \\
\text{r(B)} & \text{r(B)} & \text{r(B)} & \\
\text{w(B)} & \text{w(B)} & \text{w(B)} & \\
\text{w(C)} & \text{w(C)} & \text{w(C)} & \\
\end{array}
\]

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

\[
\begin{array}{c|cc|c|c|c}
\text{Mode of lock(s) currently held by other transactions} & \text{X} & \text{N} & \text{X} & \text{N} & \text{Grant the lock?} \\
\hline
\text{X} & \text{N} & \text{N} & \text{N} & \\
\text{N} & \text{N} & \text{N} & \\
\end{array}
\]

Compatibility matrix
Basic locking is not enough

Add 1 to both A and B (preserve A = B)
lock-X(A)
Read 100
Write 100 + 1
unlock(A)

Possible schedule under locking
lock-X(B)
Read 200
Write 200 + 1
unlock(B)

Two-phase locking (2PL)

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

Problem of 2PL

- 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 most commercial DBMS (except Oracle)

Recovery

- Goal: ensure “A” (isolation) and “D” (durability) in ACID
- 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)?
- 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:**
- **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

- **Record values before and after each modification**:
  \[
  \{ T_i, X, \text{old\_value\_of\_X}, \text{new\_value\_of\_X} \}
  \]
- A transaction $T_j$ is committed when its commit log record
  \[
  \{ T_j, \text{commit} \}
  \]
  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 $B$)

$\text{read}(A, a); a = a - 100;$

$\text{write}(A, a);$  

$\text{read}(B, b); b = b + 100;$

$\text{write}(B, b);$  

$\text{commit;}$

```
Memory
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>800</td>
</tr>
<tr>
<td>$B$</td>
<td>400</td>
</tr>
</tbody>
</table>
```

```
Log
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;T_1, \text{start}&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1, A, 800, 700&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1, B, 400, 500&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;T_1, \text{commit}&gt;$</td>
<td></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

Checkpointing

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

- Fuzzy checkpointing
  - Determine $S$, the set of currently active transactions, and log
    $\{ \text{begin-checkpoint } S \}$
  - Flush all modified memory blocks 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 $\{ \text{start-checkpoint } S \}$
- 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 $\{ T, X, \text{old, new } \}$, issue $\text{write}(X, \text{new})$

  *Basically repeats history!*

---
Recovery: undo phase

- Scan log backward
  - Undo the effects of transactions in $U$
  - That is, for each log record $(T, X, old, new)$ where $T$ is in $U$, issue `write(X, old)`, and log this operation too (part of the repeating-history paradigm)
  - Log $(T, abort)$ 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 redo (backward)