Concurrency Control

Chapter 17
Conflict Serializable Schedules

- Two schedules are *conflict equivalent* if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Schedule S is *conflict serializable* if S is conflict equivalent to some serial schedule

- Every *conflict serializable* schedule is *serializable* but the reverse is not true

- *Precedence graph*: One node per Xact; edge from $T_i$ to $T_j$ if an action of $T_i$ precedes and conflicts with one of $T_j$ actions

- *Theorem*: Schedule is conflict serializable if and only if its dependency graph is acyclic
Example

- A schedule that is not conflict serializable:

<table>
<thead>
<tr>
<th></th>
<th>T1: R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), R(B), W(B)</td>
<td></td>
</tr>
</tbody>
</table>

The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
Review: Strict 2PL

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each Xact must obtain a $S$ (*shared*) lock on object before reading, and an $X$ (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - If an Xact holds an $X$ lock on an object, no other Xact can get a lock ($S$ or $X$) on that object.

- Strict 2PL allows only schedules whose precedence graph is acyclic.
Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
Example

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

Lock table:

<table>
<thead>
<tr>
<th>Object</th>
<th>X</th>
<th>S</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>T1</td>
<td>T3</td>
</tr>
<tr>
<td>B</td>
<td>T2</td>
<td></td>
<td>T1, T4</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>T3</td>
<td>T2</td>
</tr>
</tbody>
</table>

Waits-for Graph:

T1 → T2
T1 → T4
T3 → T4
T3 → T2
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits

- If a transaction re-starts, make sure it has its original timestamp
Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph
Deadlock Detection (Continued)

Example:

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

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.

- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts commit.
Optimistic CC Model

- Xacts have three phases:
  - **READ:** Xacts read from the database, but make changes to private copies of objects.
  - **VALIDATE:** Check for conflicts.
  - **WRITE:** Make local copies of changes public.
Validation

- Test conditions that are **sufficient** to ensure that no conflict occurred.
- Each Xact is assigned a numeric id.
  - Just use a **timestamp**.
- Xact ids assigned at end of READ phase, just before validation begins.
- \( \text{ReadSet}(T_i) \): Set of objects read by Xact \( T_i \).
- \( \text{WriteSet}(T_i) \): Set of objects modified by \( T_i \).
Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.
Test 2

- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty.

Does Tj read dirty data? Does Ti overwrite Tj’s writes?
Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does +
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty +
  - WriteSet(Ti) ∩ WriteSet(Tj) is empty.

Does Tj read dirty data? Does Ti overwrite Tj’s writes?
Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph.
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
Summary (Contd.)

- Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages); should not be confused with tree index locking!

- Optimistic CC aims to minimize CC overheads in an "optimistic" environment where reads are common and writes are rare.

- Optimistic CC has its own overheads however; most real systems use locking.

- SQL-92 provides different isolation levels that control the degree of concurrency.