Concurrency Control

Chapter 17
Conflict Serializable Schedules

- Recall conflicts (WR, RW, WW) were the cause of sequential inconsistency.
- Two schedules are conflict equivalent if:
  - Involve the same actions over the same transactions.
  - Every pair of conflicting actions is ordered the same way.
- A schedule is conflict serializable if it is conflict equivalent to some serializable schedule.
Example 1

- A non-serializable schedule that is also not conflict serializable:

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

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

- A serializable schedule that is not conflict serializable:

  | T1: R(A),          W(A), C |
  | T2: W(A), C        |
  | T3: W(A), C        |

- Serializable because it is equiv to
  T1, T2, T3, or T2, T1, T3

- Not *conflict serializable*, because the ordering:
  \( R_1(A), W_2(A), W_1(A), W_3(A) \)
  is not consistent with any ordering

- Importance of this distinction is that it can be proven that
  *Strict 2PL* permits only conflict serializable schedules
Review: \textit{Strict 2PL}

\begin{itemize}
\item \textbf{Strict Two-phase Locking (Strict 2PL) Protocol}:
  \begin{itemize}
  \item Each Xact must obtain a \textit{S (shared)} lock on object before reading, and an \textit{X (exclusive)} lock on object before writing.
  \item \textit{All locks held by a transaction are released when the transaction completes}
  \item If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  \end{itemize}
\item \textbf{Strict 2PL allows only schedules whose precedence graph is acyclic (a DAG)}
\end{itemize}
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 release its locks once it has performed its desired operation (R or W). A transaction cannot 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.
- Note: locks can be released before Xact completes (commit/abort), thus relaxing Strict 2PL. 2PL starts with a “growing” phase, where locks are requested followed by a “shrinking” phase, where locks are released.
View Serializability

Schedules S1 and S2 are view equivalent if:

- If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
- If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
- If Ti writes final value of A in S1, then Ti also writes final value of A in S2

Enforcing view serializability is expensive, thus mainly of theoretical interest
Lock Management

- Lock and unlock requests are handled by the lock manager.
- Lock table entry (per table, record, or index):
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking must be atomic.
- Lock upgrades: transaction that holds a shared lock can be upgraded to hold an exclusive lock.
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Relatively rare schedules lead to deadlock
- Two ways of dealing with deadlocks:
  - Deadlock detection
  - Deadlock prevention
Deadlock Detection

- Create a **waits-for graph**:
  - Nodes are transactions
  - Edge from Ti to Tj indicates Ti is waiting for Tj to release a lock
- DBMS periodically checks for cycles in the waits-for graph
- ex: T1: A = f(B), T2: B = g(C), T3: C = h(A), arriving T1,T3,T2

<table>
<thead>
<tr>
<th>T1: S(B), R(B),</th>
<th>X(A),...</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>S(C), R(C), X(B),...</td>
</tr>
<tr>
<td>T3: S(A), R(A),</td>
<td>X(C),...</td>
</tr>
</tbody>
</table>
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)…

Diagram:

- T1 -> T2
- T4 -> T3
- T1 <- T2
- T3 <- T2
- T3 -> T3
- T3 <- T3
Deadlock Prevention

- When there is high contention for locks, detection and aborting can hurt performance
- Assign priorities (e.g., 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 abort Ti
  - **Wound-wait**: If Ti has higher priority, abort Tj; otherwise Ti waits
- When Ti re-starts, it retains its original timestamp, thus moving up the priority list
Multi-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn’t have to decide!
- Data “containers” are nested:

```
Database
  ↓
Tables
  ↓
Pages
  ↓
Tuples
contains
```
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:
  - Before locking an item, Xact must set “intention locks” on all its ancestors.
  - For unlock, go from specific to general (i.e., bottom-up).
  - **SIX mode**: Like holding the S & IX locks at the same time.

<table>
<thead>
<tr>
<th>Grant request rules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>IS</td>
</tr>
<tr>
<td>IX</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>
Multiple Granularity Lock Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must first hold an IS or IX lock on the node’s.
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.
Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.

- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.

- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.
Dynamic Databases

If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:

- T1 locks all pages containing sailor records with \(\text{rating} = 1\), and finds \(\text{oldest}\) sailor (say, \(\text{age} = 71\)).
- Next, T2 inserts a new sailor; \(\text{rating} = 1, \text{age} = 96\).
- T2 also deletes oldest sailor with \(\text{rating} = 2\) (and, say, \(\text{age} = 80\)), and commits.
- T1 now locks all pages containing sailor records with \(\text{rating} = 2\), and finds \(\text{oldest}\) (say, \(\text{age} = 63\)).

No consistent DB state where T1 is “correct”!
The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with rating = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!
Index Locking

- If there is a dense index on the \textit{rating} field using Alternative (2), T1 should lock the index page containing the data entries with \textit{rating} = 1.
  - If there are no records with \textit{rating} = 1, T1 must lock the index page where such a data entry would be, if it existed!

- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with \textit{rating} = 1 are added.
Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. $age > 2*salary$.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
  - What is the predicate in the sailor example?
- In general, predicate locking has a lot of locking overhead.
Locking in B+ Trees

- How can we efficiently lock a particular leaf node?
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.
Two Useful Observations

- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be locked (in X mode), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- We can exploit these observations to design efficient locking protocols that guarantee serializability even though they violate 2PL.
A Simple Tree Locking Algorithm

- **Search**: Start at root and go down; repeatedly, S lock child then unlock parent.

- **Insert/Delete**: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is **safe**:
  - If child is safe, release all locks on ancestors.

- **Safe node**: Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.
Example

Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*
“Optimistic” 2PL

- Basic premise: Most Xacts do not contend for the same object
- Idea: Make a local modified copy, and get locks when ready to commit
- Modified Algorithm:
  - Obtain S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of Xact, make local writes global, then release all locks.
**Timestamp CC**

- **Idea:** Give each object 2 timestamps and each transaction a timestamp:
  - read-timestamp (RTS), when it was last read
  - write-timestamp (WTS), when it was last written
  - give each Xact a timestamp (TS) when it begins:

- If action $a_i$ of Xact $T_i$ conflicts with action $a_j$ of Xact $T_j$, and $TS(T_i) < TS(T_j)$, then $a_i$ must occur before $a_j$. Otherwise, abort violating Xact.
When Xact T wants to read Object O

- If $TS(T) < WTS(O)$, this violates timestamp order of $T$ w.r.t. writer of $O$.
  - So, abort $T$ and restart it with a new, larger TS. (If restarted with same TS, $T$ will fail again! Contrast use of timestamps in 2PL for ddlk prevention.)

- If $TS(T) > WTS(O)$:
  - Allow $T$ to read $O$.
  - Reset $RTS(O)$ to $\max(RTS(O), TS(T))$

- Change to $RTS(O)$ on reads must be written to disk! This and restarts represent overheads.
When Xact T wants to Write Object O

- If $\text{TS}(T) < \text{RTS}(O)$, this violates timestamp order of $T$ w.r.t. writer of $O$; abort and restart $T$.
- If $\text{TS}(T) < \text{WTS}(O)$, violates timestamp order of $T$ w.r.t. writer of $O$.
  - **Thomas Write Rule:** We can safely ignore such outdated writes; need not restart $T$! (T’s write is effectively followed by another write, with no intervening reads.) Allows some serializable but non conflict serializable schedules:
- Else, allow $T$ to write $O$.

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

Same result as T1; T2
Timestamp CC and Recoverability

- Unfortunately, unrecoverable schedules are allowed:
- Timestamp CC can be modified to allow only recoverable schedules:
  - **Buffer all writes** until writer commits (but update WTS(O) when the write is allowed.)
  - **Block readers** T (where TS(T) > WTS(O)) until writer of O commits.
- Similar to writers holding X locks until commit, but still not quite 2PL.
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.
- Naïve locking strategies may have the phantom problem.
Summary (Contd.)

- Index locking is common, and affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).

- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.

- In practice, better techniques now known; do record-level, rather than page-level locking.
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.
Summary (Contd.)

- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block Xacts, which is similar to locking.
- Multiversion Timestamp CC is a variant which ensures that read-only Xacts are never restarted; they can always read a suitable older version. Additional overhead of version maintenance.