Concurrency Control
Chapter 17

ECS 165A – Winter 2021

Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science

UC Davis
University of California
Conflict Serializable Schedules

- **Serial schedule**: Schedule that does not interleave the actions of different transactions.

- **Equivalent schedules**: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.

- **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
Example

❖ A schedule that is not conflict serializable:

T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

❖ The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
Dependency Graph

- **Dependency graph**: One node per Xact; edge from $Ti$ to $Tj$ if $Tj$ reads/writes an object last written by $Ti$.

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

T1: $R(A), W(A)$, $R(B), W(B)$

T2: $R(A), W(A), R(B), W(B)$

![Dependency graph diagram]
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.
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 (**initial values**)
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2 (**intermediate values**)
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2 (**final values**)

<table>
<thead>
<tr>
<th>T1: R(A)</th>
<th>W(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>W(A)</td>
</tr>
<tr>
<td>T3:</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1: R(A), W(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
</tr>
<tr>
<td>T3:</td>
</tr>
</tbody>
</table>
Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of 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
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 (why?)
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)
T3: 
T4: 

There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
Deadlock Detection (Continued)

Example:

T1: S(A), R(A), S(B)
T2: X(B), W(B), X(C)
T3: S(C), R(C)
T4: There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
Deadlock Detection (Continued)

Example:

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

There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock.
Deadlock Detection (Continued)

Example:

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

There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock.
Multiple-Granularity Locks

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

```
Database
contains
Tables
contains
Pages
contains
Tuples
```
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 (i.e., top-bottom).
  - For unlock, go from specific to general (i.e., bottom-up).
  - SIX mode: Like S & IX at the same time.

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></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 hold IS or IX on parent node.
  - What if Xact holds SIX on parent? S on parent?
- 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 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 \textit{rating} = 1, and finds \textit{oldest} sailor (say, \textit{age} = 71).
  - Next, T2 inserts a new sailor; \textit{rating} = 1, \textit{age} = 96.
  - T2 also deletes oldest sailor with \textit{rating} = 2 (and, say, \textit{age} = 80), and commits.
  - T1 now locks all pages containing sailor records with \textit{rating} = 2, and finds \textit{oldest} (say, \textit{age} = 63).

- There is no consistent DB state where T1 is “correct”!
The Problem

❖ T1 implicitly assumes that it has locked the set of all sailor records with \textit{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 rating field using Alternative (2), T1 should lock the index page containing the data entries with rating = 1.
  - If there are no records with 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 rating = 1 are added.
Predicate Locking

❖ Grant lock on all records that satisfy some logical predicate, e.g. \( \text{age} > 2 \times \text{salary} \).

❖ 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.
Optimistic CC (Kung-Robinson)

- Locking is a conservative (pessimistic) 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.
Kung-Robinson 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.
Kung-Robinson 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.
Kung-Robinson 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. (Why then?)
- \( \text{ReadSet}(Ti) \): Set of objects read by Xact Ti.
- \( \text{WriteSet}(Ti) \): Set of objects modified by Ti.
Test 1

❖ For all $i$ and $j$ such that $T_i < T_j$, check that $T_i$ completes before $T_j$ begins.
Test 2

❖ For all i and j such that Ti < Tj, check that:
   ▪ Ti completes before Tj begins its Write phase +
   ▪ WriteSet(Ti) \( \cap \) 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) \(\bigcap\) ReadSet(Tj) is empty +
  - WriteSet(Ti) \(\bigcap\) WriteSet(Tj) is empty.

Does Tj read dirty data? Does Ti overwrite Tj’s writes?
Comments on Serial Validation

- Assignment of Xact id, validation, and the Write phase are inside a **critical section**!
  - I.e., Nothing else goes on concurrently.
  - If Write phase is long, major drawback.

- Optimization for Read-only Xacts:
  - Don’t need critical section (because there is no Write phase).
Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes "global".
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.
``Optimistic” 2PL (analogous to 2VCC)

- If desired, we can do the following:
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of Xact, make writes global, then release all locks.

- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in Xacts being blocked, waiting for locks.
  - However, no validation phase, no restarts (modulo deadlocks).
Timestamp CC

- **Idea:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), 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, restart 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 deadlock 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 $TS(T) < RTS(O)$, this violates timestamp order of T w.r.t. writer of O; abort and restart T.
- If $TS(T) < WTS(O)$, 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>T1</th>
<th></th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
<td>W(A) Commit</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
Timestamp CC and Recoverability

- Unfortunately, unrecoverable schedules are allowed:
  - Read is aborted if \( \text{TS}(T) < \text{WTS}(O) \)
  - Write is aborted if \( \text{TS}(T) < \text{RTS}(O) \) or \( \text{TS}(T) < \text{WTS}(O) \)

- Timestamp CC can be modified to allow only recoverable schedules (*any similarity to 2VCC?*):
  - **Buffer all writes** until writer commits (but update WTS(O) when the write is allowed.)
  - **Block readers** \( T \) (where \( \text{TS}(T) > \text{WTS}(O) \)) until writer of \( O \) commits.

- Similar to writers holding X locks until commit, but still not quite 2PL.
Multiversion Timestamp CC
(Any Similarity to L-Store?)

Idea: Let writers make a “new” copy while readers use an appropriate “old” copy:

MAIN SEGMENT (Current versions of DB objects)
**Multiversion Timestamp CC**

*(Any Similarity to L-Store?)*

**Idea:** Let writers make a “new” copy while readers use an appropriate “old” copy:

- **MAIN SEGMENT** (Current versions of DB objects)
- **VERSION POOL** (Older versions that may be useful for some active readers.)
**Multiversion Timestamp CC**

*(Any Similarity to L-Store?)*

**Idea:** Let writers make a “new” copy while readers use an appropriate “old” copy:

- **MAIN SEGMENT** (Current versions of DB objects)
  - O

- **VERSION POOL** (Older versions that may be useful for some active readers)
  - O’
  - O”

Readers are always allowed to proceed.
But may be blocked until writer commits.
Multiversion CC (Contd.)

- Each version of an object has its writer’s TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are “too old to be of interest”.
- Each Xact is classified as Reader or Writer.
  - Writer may write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.
Reader Xact

- For each object to be read:
  - Finds **newest version** with $WTS < TS(T)$. (Starts with current version in the main segment and chains backward through earlier versions.)

- Assuming that some version of every object exists from the beginning of time, **Reader Xacts are never restarted.**
  - However, might block until writer of the appropriate version commits.
**Writer Xact**

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version** $V$ s.t. $WTS < TS(T)$.
  - If $RTS(V) < TS(T)$,
    - $T$ makes a copy $CV$ of $V$, with a pointer to $V$, with $WTS(CV) = TS(T)$, $RTS(CV) = TS(T)$.
    - Write is buffered until $T$ commits; other Xacts can see $TS$ values but can’t read version $CV$.
  - Else, reject write.
### Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable Reads</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
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.
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).

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