Concurrent Control
R & G - Chapter 19

Smile, it is the key that fits the lock of everybody's heart.
Anthony J. D'Angelo, The College Blue Book

Review
- DBMSs support concurrency, crash recovery with:
  - ACID Transactions
  - Log of operations
- A serial execution of transactions is safe but slow
  - Try to find schedules equivalent to serial execution
- One solution for serializable schedules is 2PL

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

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)

  Dependency graph
  - 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 an operation of Ti conflicts with an operation of Tj and Ti's operation appears earlier in the schedule than the conflicting operation of Tj.
- **Theorem**: Schedule is conflict serializable if and only if its dependency graph is acyclic

An Aside: 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

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

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

  View serializability is "weaker" than conflict serializability!
  - Every conflict serializable schedule is view serializable, but not vice versa!
  - I.e. admits more legal schedules
App-Specific Serializability

- In some cases, application logic can deal with apparent conflicts
  - E.g. when all writes commute
  - E.g. increment/decrement (a.k.a. "escrow transactions")

| T1: x=R(A), W(A=x+1), z=R(A), W(z=z+1) | T2: y=R(A), W(A=y-1) |

- Note: doesn’t work in some cases for (American) bank accounts
  - Account cannot go below $0.00!!
- In general, this kind of app logic is not known to DBMS
  - Only sees encapsulated R/W requests
  - But keep in mind that general serializability is "weaker" than even view serializability

Review: Strict 2PL

<table>
<thead>
<tr>
<th>Lock Compatibility Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

- **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 a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- **Can result in Cascading Aborts!**
  - STRICT (!) 2PL "Avoids Cascading Aborts" (ACA)

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
  - Requires latches ("semaphores"), which ensure that the process is not interrupted while managing lock table entries
  - See CS162 for implementations of semaphores
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems

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 gets 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), S(D), S(B)
T2: X(B), X(C)
T3: S(D), S(C), X(A)
T4: X(B), S(A), X(B)

Deadlock Detection (cont.)

• In practice, most systems do detection
  - Experiments show that most waits-for cycles are length 2 or 3
  - Hence few transactions need to be aborted
  - Implementations can vary
    • Can construct the graph and periodically look for cycles
    • Can do a "time-out" scheme: if you’ve been waiting on a lock for a long time, assume you’re deadlock and abort

Summary

• Correctness criterion for isolation is "serializability".
  - In practice, we use "conflict serializability", which is somewhat more restrictive but easy to enforce.
• There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Locks directly implement the notions of conflict.
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.

Things We’re Glossing Over

• What should we lock?
  - We assume tuples here, but that can be expensive!
  - If we do table locks, that’s too conservative
    - Multi-granularity-locking
• Locking in indexes
  - don’t want to lock a B-tree root for a whole transaction!
  - actually do non-2PL "latches" in B-trees
• CC w/out locking
  - "optimistic" concurrency control
  - "timestamp" and multi-version concurrency control
    - locking usually better, though
• App-specific tricks
  - e.g. increment/decrement ("escrow transactions")

In case we have time

• The following is an interesting problem
• We will not discuss how to solve it, though!
Dynamic Databases – The “Phantom” Problem

- If we relax the assumption that the DB is a fixed collection of objects, even strict 2PL (on individual items) will not assure serializability:
  - Consider T1 – “Find oldest sailor for rating 1”
    - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71).
    - May find these pages via a clustered index on rating, and never touch (or lock) any other pages
    - Next, T2 inserts a new sailor; rating = 1, age = 96.
    - T2 commits.
    - T1 issues another query to find the oldest sailor for rating 1.
      - A phantom sailor appears! (and she’s 96 years old!)
  - No serial execution where T1’s result could happen!

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!
  - e.g. table locks

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.
  - too expensive!
  - Fancier index locking tricks are used in practice