Goals for Today

• Understand how to manage locks and deadlock when implementing 2PL or Strict 2PL

• Reason about issues that can arise when data is inserted or deleted

• Discuss Isolation levels in a DBMS

Yes S1 and S2 are conflict-equivalent.
No neither is conflict serializable.

Two-Phase Locking (2PL)

1) Protocol: each transaction must
   – Get a S (shared) or an X (exclusive) lock on object before reading
   – Get an X (exclusive) lock on object before writing

   Can upgrade a Shared lock to an Exclusive lock! (when okay?)

2) All lock requests precede all unlock requests!
   I.e., a transaction can not request additional locks once it releases any locks

Growing Phase

Shrinking Phase

# Locks Held

Time
Basic Locking: Example (Take 2-abort)

A = 1000, B = 2000, Output = ?

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock_X(A)</strong> &lt;granted&gt;</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>A = A-50</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td><strong>Lock_X(B)</strong> &lt;granted&gt;</td>
<td>&lt;granted&gt;</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>Read(B)</td>
</tr>
<tr>
<td><strong>Lock_S(B)</strong></td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B = B +50</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>PRINT(A+B)</td>
</tr>
</tbody>
</table>

Avoiding Cascading Aborts: Strict 2PL

- Problem with 2PL: cascading aborts
- Another example: rollback of T1 requires rollback of T2
  - T1: R(A), W(A), R(B), W(B), Abort
  - T2: R(A), W(A)

- Solution: **Strict Two-phase Locking** (Strict 2PL):
  - Same as 2PL, except:
  - **All locks held by a transaction are released only when the transaction completes**

  *Consequence:* a writer will block all other readers until the writer commits or aborts

Exercise 2

a) Yes 2PL, No Strict 2PL

b) Neither (schedule not conflict-serializable)

View Serializability

- Schedules S1 and S2 are **view equivalent** if:
  - If T1 reads initial value of A in S1, then T1 also reads initial value of A in S2
  - If T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2
  - If T1 writes final value of A in S1, then T1 also writes final value of A in S2

Checking for this is NP-complete!
Lock Management

- Lock/unlock requests are handled by the Lock Manager
  - Have table with entry for each currently held lock

- What object is being locked?
  - Possibilities: table(s), row(s), page(s)...
  - Too coarse-grained limits concurrency!
  - More on granularity later...

- Lock table entry
  - Object id of object being locked (e.g., table, row, page)
  - (Pointer to) list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - (Pointer to) queue of lock requests

Try Exercise 3

<table>
<thead>
<tr>
<th>ObjectID</th>
<th>LockType</th>
<th>Xacts</th>
<th>XactsRequesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>S</td>
<td>T1, T3</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>T2</td>
<td>T1, T4</td>
</tr>
<tr>
<td>C</td>
<td>S</td>
<td>T3</td>
<td>T2</td>
</tr>
</tbody>
</table>

Lock Management (cntd)

- When a lock request arrives
  - Check if any xact currently holds a conflicting lock on the object
  - If not, create an entry and grant the lock
  - Else, put the requesting xact on the wait queue

*Locking and unlocking have to be atomic operations!*

Basic Locking: Example (Take 3)

<table>
<thead>
<tr>
<th>Lock_X(A) &lt;granted&gt;</th>
<th>Lock_S(B) &lt;granted&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

- Can see cycle in 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

- Two main ways of dealing with deadlocks in DBMS:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps

- Suppose Ti wants a lock that Tj holds
  - Two possible policies:
    - **Wait-Die**: If Ti is older, Ti waits for Tj; otherwise Ti aborts
    - **Wound-wait**: If Ti is older, Tj aborts (gets “wounded”); otherwise Ti waits

  *In both, the older never aborts*

- If a transaction re-starts, make sure it gets its original timestamp

Why?

Deadlock Exercise: 4(a)

- Recall **Wait-Die**:
  - Ti wants a lock that Tj holds
  - If Ti is older, Ti waits for Tj; otherwise Ti aborts

Schedule 1:
1) T1 requests for and gets S(A)
2) T2 request X(A), can’t get it. Aborts because T2 younger than T1
3) T3 requests and gets X(B)
4) T1 requests X(B), can’t get it. Waits because T1 older than T3
5) T3 commits, releases lock on B
6) T1 gets X(B), then commits
   (T2 restarted)

Schedule 2:
(same as schedule 1, except T2 makes a bit more progress before aborting)

Deadlock Detection

- Alternative is to allow deadlocks to happen but to check for them and fix them if found.

- Periodically check for cycles in the waits-for graph

- If cycle detected – find a transaction whose removal will break the cycle and kill it
Deadlock Detection (Cntd)

Example:

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

Deadlock Exercise: 4(b)

Schedule 1:
T2 blocks on T1 on object A
T1 blocks on T3 on object B
When T3 finishes, T1 resumes and gets B
When T1 finishes, T2 resumes and gets A (and then B)

Schedule 2:
T2 blocks on T1 on object A
T3 blocks on T2 on object B
T1 blocks on T2 on object B
DEADLOCK! Waits-for-graph has cycle between T1 and T2

“Dynamic” Databases

• Relax assumption that database is a static set of objects

• With Insert and Delete possible, even Strict 2PL (on individual objects) will not assure serializability

The “Phantom” Problem – Example 1

• Consider T1 – “Find oldest sailor”
  T1 locks all Sailor records,
  finds oldest sailor (age = 71)
  T2 inserts a new sailor; age = 96
  commits

  T1 checks for the oldest sailor,
  finds oldest sailor (age = 96)
  No serial execution where T1’s result could happen!

• The sailor with age 96 is a “phantom tuple” from T1’s point of view --- first it’s not there, then it is
The “Phantom” Problem – Example 2

• Consider T3 – “Find oldest sailor for each rating”

T3 locks all pages containing sailor records with rating = 1
finds oldest sailor (age = 71)

T4 inserts a new sailor; rating = 1, age = 96
T4 also deletes oldest sailor with rating = 2, age = 80
commits

T3 now locks all pages containing sailor records with rating = 2, and finds oldest (age = 63).

• T3 saw only part of T4’s effects!

No serial execution where T3’s result could happen!

The Problem

• How do you lock something that does not yet exist??

• T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.
  – Assumption only holds if no sailor records are added while they are executing!
  – Need some mechanism to enforce this assumption, e.g., index locking

• Conflict serializability on reads and writes of individual objects guarantees serializability only if the set of objects is fixed

Isolation Levels in SQL Standard

• Idea: give users control over locking overhead incurred by their xacts
• Xacts can be specified with desired Isolation Level
  – Also, access mode like “read-only” only gets S locks

<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>No</td>
<td>No</td>
<td>Does not get read locks, (not allowed to write objects)</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>No</td>
<td>Write locks held to commit. Get read locks, but release those right away</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>Strict 2PL. Locks before read &amp; write, on individual objects</td>
</tr>
</tbody>
</table>

Does not get read locks, (not allowed to write objects)
Write locks held to commit. Get read locks, but release those right away
Maybe
Strict 2PL. Locks before read & write, on individual objects
Strict 2PL. Gets locks before read/write, including on sets of objects (index locks)